UNIVERSIDADE FEDERAL DE SANTA CATARINA

PROGRAMA DE PÓS-GRADUAÇÃO EM

CIÊNCIA DA COMPUTAÇÃO

Edmilson José Molinari

PADRÕES DE PROJETO PARA DESENVOLVER APLICAÇÕES DISTRIBUÍDAS COM CORBA E JAVA

Dissertação submetida à Universidade Federal de Santa Catarina como parte dos requisitos para a obtenção do grau de Mestre em Ciência da Computação

Orientador: Prof. Dr. João Bosco Mangueira Sobral

Florianópolis, Janeiro/2002.

PADRÕES DE PROJETO PARA

DESENVOLVER APLICAÇÕES

DISTRIBUÍDAS COM CORBA E JAVA

Esta dissertação foi julgada adequada para a obtenção do título de Mestre em Ciência da Computação Área de Concentração Sistemas de Computação e aprovada em sua forma final pelo Programa de Pós-Graduação em Ciência da Computação.

_____

Prof. Fernando Álvaro Ostuni Gauthier, Dr.

Banca Examinadora:

_____

Prof. João Bosco Mangueira Sobral, Dr.

_____

Prof. Rosvelter João Coelho da Costa, Dr.

_____

Prof. Paulo Sérgio da Silva Borges, Dr.

_____

Prof. Vitório Bruno Mazzola, Dr.

Agradecimentos

A Deus, por permitir a conclusão de tão importante empreendimento.

Agradeço mui especialmente ao Prof. Dr. João Bosco Mangueira Sobral pela confiança, apoio, estímulo e dedicação.

A Rodrigo Bianco, pelo auxílio dado e pelas discussões que tiveram influência neste trabalho.

A minha amada esposa Emilia da Natividade Domingues e aos meus queridos filhos Bruna e Rafael que me apoiaram ao longo de todo o mestrado.

Aos meus pais, irmãos e cunhado por darem o apoio necessário nesta caminhada difícil.

Aos meus amigos Marcus, Geber e Evandro que auxiliaram-me para a conclusão deste trabalho.

Sumário

Resumo

O aparecimento do desenvolvimento de software orientado a objeto e computação cliente/servidor melhorou a qualidade do processo de desenvolvimento. Entretanto, não foi suficiente para erradicar os problemas existentes nessa relação, principalmente aqueles referentes à reutilização, incompatibilidade entre sistemas operacionais e linguagens de programação. Esses problemas tem nos motivado a realizar este trabalho, no qual busca-se contribuir com o desenvolvimento de projetos de software orientado a objeto. Empregando o mecanismo da reutilização para a implementação de sistemas distribuídos, portabilidade e interoperabilidade são providos para sistemas legados. Este trabalho mostra a elaboração de um catálogo de padrões de projeto. Esses padrões oferecem as bases para o desenvolvimento de aplicações distribuídas com CORBA e Java. Eles proporcionam através de classes, a construção de novos sistemas.

Palavras-Chave: padrões de projeto, reutilização, interoperabilidade, portabilidade.

Abstract

The appearance of object-oriented development technology and client/server computation has improved the quality and dynamism of software development. However, it was not enough to eradicate the existing problems in this relationship, especially those regarding to reusability, incompatibility among the operational systems and programming languages. These problems have motivated us to perform this study, which we tried to contribute with the development of software object oriented projects. By using the reusability mechanism in heterogeneous distributed systems, portability and interoperability are provided to legacy systems through Web applications. This work resulted in the elaboration of a design pattern catalog. These patterns offer the base for the development of distributed aplications with CORBA and Java. They providing through classes to understand legacy systems or they benefit the building of new systems.

Keyword: design patterns; reusability; interoperability; portability;.

Índice de Figuras

1. INTRODUÇÃO

Este trabalho focaliza uma parte da Engenharia de Software, voltada ao uso de padrões de projeto (*design patterns*) que constituem declarações de problemas e soluções que detalham abordagens de senso comum predefinidas para resolver um problema de projeto [MOWB97]. O propósito principal é criar um catálogo de padrões de projeto apropriados para o desenvolvimento de aplicações baseadas na Web, utilizando a padronização CORBA [OMG97] para objetos distribuídos e a linguagem de programação Java [BILL98]. Os padrões de projeto deste catálogo são direcionados para qualquer domínio de aplicação, mas tendo como objetivo principal, as aplicações cliente/servidor orientadas a objetos, distribuídas via Web. Embora existem muitas tecnologias úteis voltadas para a Web, existem poucos exemplos de soluções Internet que possamos chamá-las de verdadeiros *design patterns* [MOWB97].

Tendo-se como enfoque fundamental o paradigma orientado a objetos distribuídos, existe a necessidade de se trabalhar com especificações de requisitos complexos para a construção destes sistemas [ORFA99] e assim questões críticas no projeto de um sistema surgem precisando serem solucionadas. Algumas soluções boas e testadas para os problemas de projeto podem ser expressas como um conjunto de princípios e padrões. Esses padrões constituem fórmulas nomeadas de soluções de problemas que codificam o conhecimento, os idiomas e os princípios existentes de projetos exemplares e quanto mais apurado e amplamente usado, melhor [LARM99]. Esta dissertação, ao trabalhar com padrões, pretende favorecer o uso, com habilidade, destes, para o projeto orientado a objetos.

Com isto estabelecemos algumas soluções apropriadas para trabalhar com arquitetura cliente/servidor em duas, três ou várias camadas na Web, possibilitando aplicações orientadas a objetos serem desenvolvidas distribuídas em ambiente heterogêneo ou não.

Orientação a objetos, sejam eles distribuídos ou não, é a tecnologia que por conta de sua própria estrutura vem permitindo uma das suas maiores promessas que é a reutilização de processo incorporando os conceitos de flexibilidade, extensibilidade e

portabilidade, no sentido de gerar uma grande economia de custos e aumento de produtividade. Os padrões de projeto se encaixam neste contexto, tendo como finalidade primordial a reutilização de código parcialmente desenvolvido e testado. Desta forma, uma meta importante a ser considerada para o aumento de produtividade e redução de custos envolvidos na produção de software, é a reutilização de código, uma técnica importante neste contexto, uma vez que permite a construção de software a partir de modelos (*templates*) bem especificados, semi-implementados e testados. Pretende-se assim, a melhoria da qualidade do software de desenvolvimento, quanto a documentação, e na qualidade dos produtos de software desenvolvidos.

De acordo com [BELL01] existem algumas condições básicas sem as quais não haverá reutilização:

- Que os objetos sejam criados especificamente visando a reutilização;
- Que os desenvolvedores estejam cientes dos objetos reutilizáveis existentes;
- Que seja fácil aos desenvolvedores encontrar os objetos de que necessitam durante o desenvolvimento;
- Que o processo de software preveja explicitamente pontos em que a reutilização deve ser considerada;
- Que se utilize uma ferramenta adequada, com conhecimento para o desenvolvimento proposto.

Um componente a ser reutilizado precisa ser melhor construído, testado e documentado, do que um outro que esteja destinado a ser utilizado uma única vez.

A heterogeneidade e distribuição de objetos em servidores de aplicações cada vez mais se tornam um princípio fundamental para a flexibilidade e longa vida da aplicação. E, durante esta última década, o Object Management Group (OMG) tem garantido que se possa integrar o que já está construído àquilo que se está construindo e que se irá construir, utilizando o Common Object Request Broker Architecture (CORBA) uma arquitetura para a integração entre aplicações orientadas a objetos ou não e que

permita implementações em diferentes linguagens e em diferentes plataformas computacionais.

À medida que a inovação tecnológica continua acelerada, a demanda por integração de aplicações cresce, sendo necessária uma tecnologia que coloque a interoperabilidade no centro da sua infra-estrutura. Sabemos que nunca haverá um único sistema operacional ou uma única arquitetura de rede que possa substituir tudo aquilo que vimos utilizando há anos. Por isso, a necessidade de escolha de produtos que venham trabalhar com interoperabilidade.

As aplicações distribuídas têm várias vantagens em relação aos sistemas monolíticos: as formas têm escalabilidade; tolerância à falhas e promove recursos compartilhados. Contudo, aplicações distribuídas introduzem preocupações do tipo: tráfego na rede; escalabilidade de servidor e aplicação cliente/servidor via Web. Isto pode significar fracasso de projeto se negligenciado. Neste contexto, o projeto e teste de aplicações distribuídas são considerados mais difíceis que nos sistemas monolíticos.

Padrões de projeto para aplicação distribuída utilizando CORBA e Java vem mostrar como refinar e minimizar a complexidade inerente a tais aplicações. A documentação contendo informações do contexto, problema, solução e exemplos de códigos fontes, faz com que o projeto ou partes de uma aplicação possam ser reutilizadas.

Para tanto estabelecemos um catálogo de padrões de projeto que se aplicam ao desenvolvimento de aplicações com objetos distribuídos que podem viabilizar a arquitetura cliente/servidor através de objetos na Web.

Padrões de projeto podem não ser o único fator preponderante nesta história, mas queremos mostrar que, se bem trabalhados, podem trazer resultados expressivos para a melhora na qualidade de software de qualquer empresa, seja ela grande ou pequena.

Para a implementação de tais padrões, utilizaremos a linguagem de programação Java, que apresenta características que viabilizam a implementação do projeto. Programas em Java podem ser desenvolvidos orientados a objetos, possuindo portabilidade que nenhuma outra linguagem oferece e são de fácil integração com o ambiente WEB.

Faz parte, também, no contexto deste trabalho, a Unified Modeling Language (UML), uma notação padrão para a modelagem e criação de artefatos de software orientado a objetos, adotado como padrão em 1997 pelo OMG. Hoje, no mercado, já existem algumas ferramentas como TogetherSoft [TOGE02] e Rational Rose [RATI02], que oferecem métodos de modelagem de análise e projeto orientados a objetos utilizando a notação da UML para a criação e modelagem de artefatos de software utilizando padrões de projeto.

Para demonstrar a utilização do catálogo de padrões e mostrar sua viabilidade, utilizamos um sistema comercial denominado neste trabalho de Balcão. Através do catálogo de padrões de projeto aqui proposto, enfatizamos que documentar padrões e utilizá-los beneficiam o desenvolvimento de novos sistemas e atualização de sistemas legados, mediante a colocação dos princípios de orientação a objetos em evidência e melhorando os aspectos flexibilidade, extensibilidade, portabilidade, e reutilização.

No que segue, no capítulo 2 apresentaremos os conceitos de computação distribuída, como o CORBA. No capítulo 3 são apresentados os conceitos de padrões de projeto, as linguagens de padrões de Buschmann, Gamma e Mombray e seus respectivos modelos. O catálogo de padrões de projeto proposto por esta dissertação faz parte do capítulo 4. No capítulo 5, aplica-se o catálogo de padrões em um estudo de caso de projeto de uma aplicação Balcão que tem como objetivo mostrar a eficiência e reutilização do catálogo. E, no capítulo 6, será apresentada a conclusão e os trabalhos futuros.

2. OBJETOS DISTRIBUÍDOS

Vários computadores pessoais e estações de trabalho (workstations) tomaram conta dos ambientes de trabalho de todas as corporações e a integração entre estes computadores buscando o processamento distribuído é uma necessidade. Os sistemas que devem ser executados nestes ambientes também têm características diferenciadas, e, para atender a estas novas características, o uso da tecnologia de orientação a objetos está passando a fazer parte do processo de desenvolvimento de software de todas as corporações.

A nova realidade presente em grande parte das empresas atuais, que estão usando sistemas distribuídos e orientação a objetos , dá origem à área dos objetos distribuídos [MAIN98]. Modelos de objetos distribuídos serão analisados, mas será dado enfoque ao modelo CORBA da OMG. A arquitetura OMA e os componentes CORBA são descritos mais detalhadamente em outros tópicos. Para implementar as classes e objetos, será utilizada a linguagem de programação Java, tendo em vista a mesma oferecer transparência de implementação, portabilidade de aplicações independentes de plataforma e um enorme potencial para trabalhos direcionados para a internet.

Conforme Robert Orfali, Dan Harkey e Heri Edwards em [ORFA96], [ORFA97] e [ORFA95], a indústria de objetos distribuídos estará se transformando no principal mercado da computação, ultrapassando a indústria de bancos de dados nos próximos anos, conforme mostra o gráfico na figura 1 descrito em [OMG97].

Objetos distribuídos começaram a sua trajetória através do *middleware* (é um software que reside entre a aplicação e as operações internas do sistema da aplicação, isolando-as dos detalhes de baixo nível e das complexidades que suportam os sistemas [OTTE96]), mas hoje está invadindo todas as áreas, inclusive a internet.

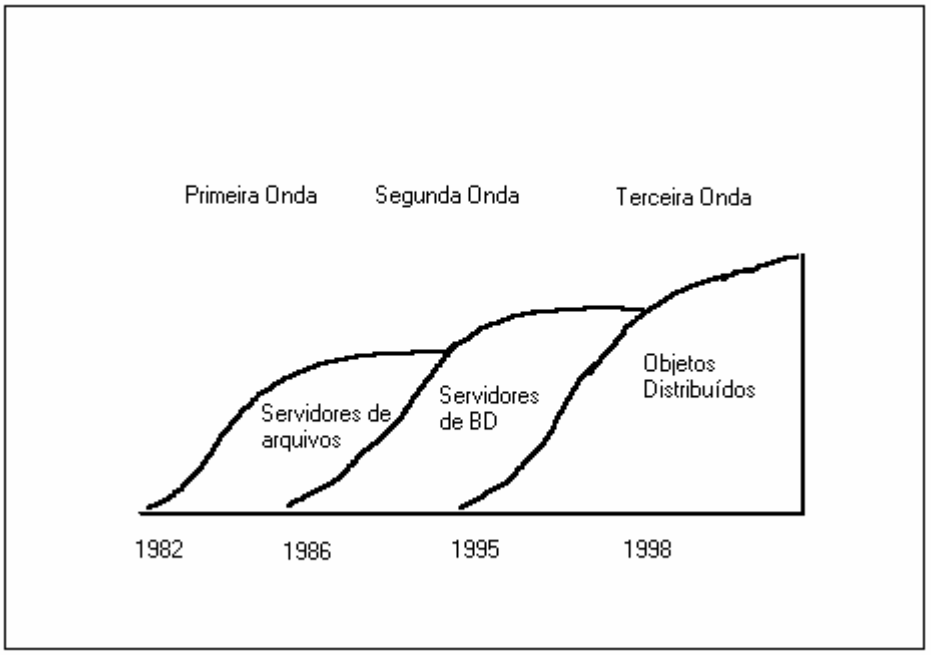

Figura 1 – As “ondas” de cliente/servidor.

2.1. Computação Distribuída

Para se entender melhor a conceituação do que é um sistema distribuído, é necessário partir do significado de um sistema de rede.

Consiste o sistema de rede em um conjunto de computadores autônomos com sistemas operacionais independentes conectados, formando uma rede. Não existe um sistema-mestre único. Em um sistema de rede, as interações entre os sistemas individuais são essencialmente para transferir arquivos, usando sessões ponto-a-ponto que podem ser longas e duradouras. A interação é mais em forma de comunicação do que próxima à idéia de cooperação.

Sistemas Distribuídos são, essencialmente, uma extensão de sistemas de redes, onde a interação abrange a *comunicação* e *cooperação.*

A computação distribuída pode ser definida como duas ou mais partes de software compartilhando informações entre si, podendo estas rodar em uma mesma máquina ou em máquinas diferentes conectadas em rede [COUL95].

O crescimento e a popularização das redes de computadores observado nos últimos anos, tanto em número quanto em tamanho e complexidade, vieram acompanhadas da necessidade de interconectar redes anteriormente isoladas para permitir que computadores em diferentes redes compartilhem informações e trabalhem em cooperação.

Com o intuito de propor padrões para sistemas abertos com arquitetura de suporte ao desenvolvimento de aplicações distribuídas em ambientes heterogêneos, várias empresas tem trabalhado para impor um modelo para este ambiente e, para que tal modelo seja aceito pelas organizações internacionais e pelo mercado (padrão de fato). Dentre as arquiteturas projetadas, até o momento, temos: DCE (*Distributes Computing Environment*), ANSA (*Advanced Network System Architecture*), CORBA (*Common Object Request Broker Architecture*), OLE?COM, *OpenDoc*, DCOM e JAVA. Em ordem cronológica, DCE e ANSA são as tecnologias mais antigas, sendo que as mais atuais utilizaram a maior parte dos conceitos e modelos de ambas para implementar suas novas propostas.

2.2. Cliente/Servidor

Com a evolução da Tecnologia de Informação surge a computação cliente/servidor, modelo de gerenciamento de informação que divide o processamento de informações entre um computador que requer um serviço e outro que devolve o serviço.

Com este modelo, os componentes da aplicação dividem-se da seguinte maneira: os componentes da base de dados ficam no servidor; os componentes da interface do usuário ficam no cliente e os das regras de negócios em um ou no outro, ou em ambos.

Quando é feita a mudança de um componente da parte do cliente, nova cópia do componente cliente (executável ou pacotes executáveis) tem de ser distribuída para cada usuário. Com o avanço de multicamadas no modelo cliente/servidor, o modelo original ficou conhecido como arquitetura de duas camadas (Two-tier), conforme figura 2.

A arquitetura cliente/servidor foi, sob vários aspectos, uma revolução na computação. Porém, apesar de resolver os problemas das aplicações baseadas em Mainframes, a arquitetura cliente/servidor não estava sem falhas. Por exemplo, a funcionalidade de acesso ao Banco de Dados (tal como consulta(Query)) e as regras de negócios, estavam embutidas no componente Cliente, e qualquer alteração nas regras de negócios, ou no acesso ao Banco de Dados, ou mesmo no próprio banco, freqüentemente obrigavam a atualização do componente para todos os usuários, resultado em uma redistribuição da aplicação.

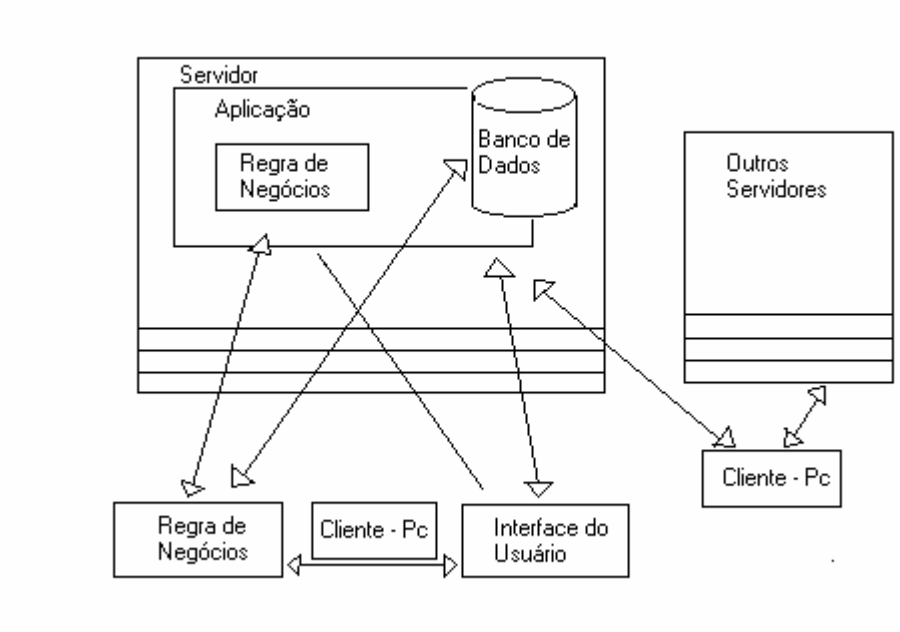

Figura 2 - Arquitetura duas camadas cliente/servidor.

O problema com a tradicional arquitetura cliente/servidor (duas camadas, “two-tier”) foi substituído através das múltiplas camadas. Conceitualmente, uma aplicação pode ter qualquer número de camadas, mas a mais popular das arquiteturas multicamadas

é a “tree-tier”, a qual particiona o sistema em três camadas lógicas: a camada de interface de usuário, a de regras de negócio, e a de acesso ao Banco de Dados conforme figura 3.

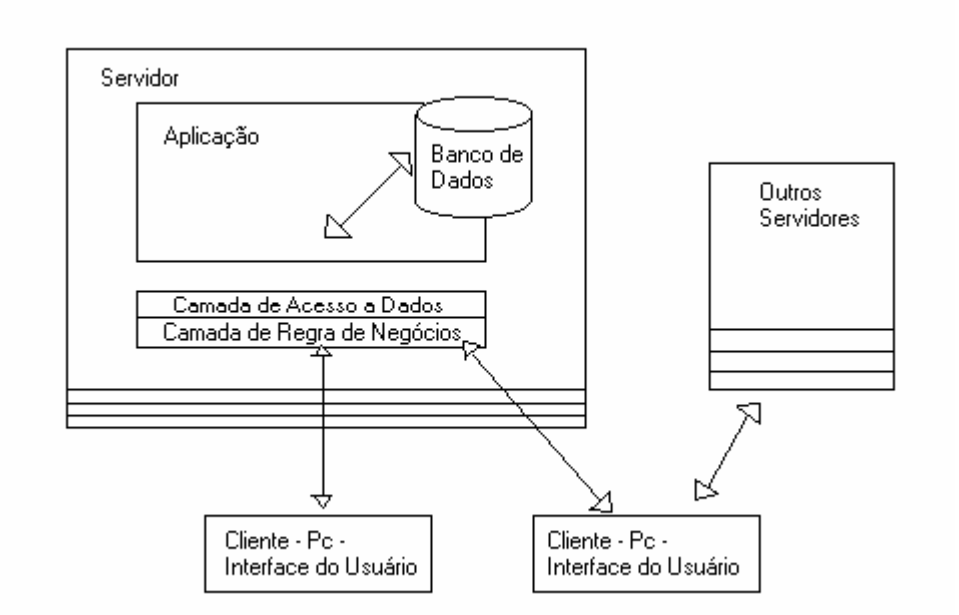

Figura 3 - Arquitetura Três Camadas cliente/servidor.

A arquitetura cliente/servidor multicamadas adiciona à arquitetura de duas camadas importantes pontos. Talvez o mais importante seja o fato de tornar a aplicação menos frágil, isolando as alterações do cliente do restante da aplicação e também, pelo fato dos componentes executáveis serem mais refinados, permitindo uma maior flexibilidade no desenvolvimento de qualquer aplicação.

A arquitetura cliente/servidor multicamadas reduz a fragilidade da aplicação, fornecendo maior isolamento entre as camadas. A camada de interface do usuário comunica-se somente com a das regras de negócios e nunca diretamente com a de acesso a dados. A camada de regra de negócio, por sua vez, comunica-se por um lado com a camada de interface de usuário e por outro com a camada de acesso a dados. Deste modo, alterações feitas na camada de acesso a dados não afetarão a camada de interface com usuário, pois são isoladas uma da outra. Esta arquitetura permite que as alterações sejam feitas na aplicação com menor probabilidade de afetar o componente Cliente (o qual, lembre-se, tem de ser redistribuído sempre que for alterado).

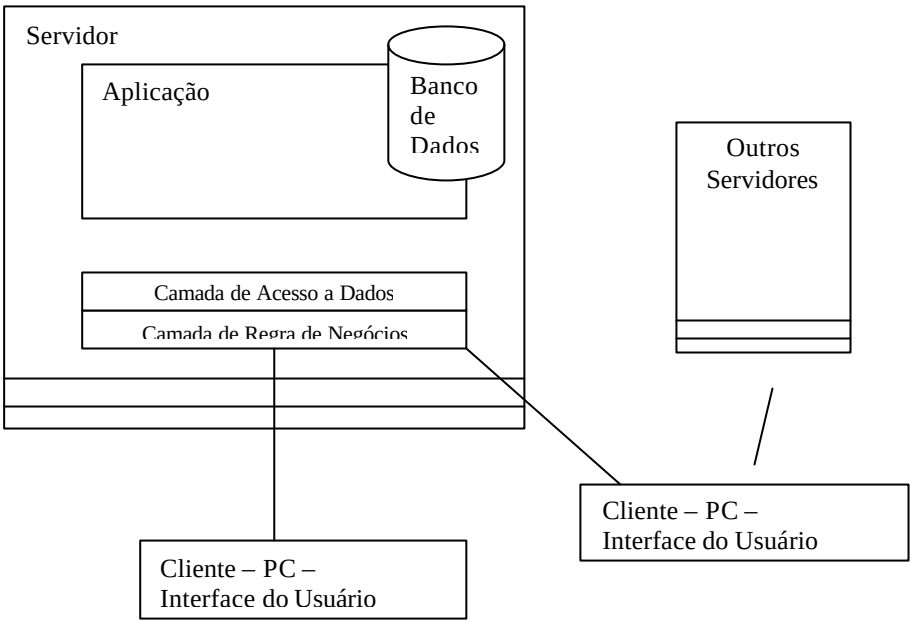

Figura 4 - Arquitetura multicamadas.

Uma outra vantagem da arquitetura multicamadas é que os componentes de regra de negócios, de acesso a dados e o próprio Banco de Dados podem rodar em máquinas diferentes, distribuindo melhor a carga das aplicações e tornando-as mais robustas e escaláveis, isto é, não haverá a distinção entre cliente e servidor, sendo que o cliente pode fornecer um componente, tal como o servidor [RICC00].

2.3. Tecnologias de Computação Distribuída

A computação distribuída pode ser definida como duas ou mais partes de software compartilhando informações entre si, podendo estas estarem rodando em uma mesma máquina ou em máquinas diferentes conectadas em rede [COUL95]. Um sistema de computação distribuída possui um processo cliente e um processo servidor que são executados em computadores que se comunicam entre si utilizando algum protocolo de

rede. Em sistemas distribuídos, os clientes pedem para os servidores executarem algumas funções, caracterizando assim o modelo cliente/servidor.

Tecnologias de sistemas distribuídos baseados no modelo cliente/servidor proporcionam algumas facilidades extras como, por exemplo, a comunicação assíncrona. A comunicação síncrona ocorre quando uma parte do software envia uma mensagem para uma outra parte do software e então espera pela resposta. Dá-se a comunicação assíncrona quando uma parte do software envia uma mensagem para uma outra parte e continua a operar, esperando que a resposta retorne em algum instante.

Quando se deseja utilizar e implementar aplicações distribuídas, certamente o CORBA não é a única alternativa, existem outros mecanismos de utilização que, dependendo do tipo e da natureza da aplicação, devem ser considerados (plataforma operacional, linguagem de programação). A seguir, são descritas resumidamente as principais tecnologias para a implementação de sistemas distribuídos [RICC00]:

- Socket Programming. O socket é um canal de comunicação através do qual uma aplicação se comunica com outra. Porém, pela sua simplicidade não, é bem aplicada na manipulação de tipo de dados ou de aplicações complexas;

- RPC ou RFC (Remote Procedure ou Function Call). É uma evolução do Socket Programming, sendo de fácil implementação, possuindo um desempenho muito interessante. É um mecanismo que permite que clientes façam chamadas de procedimentos em servidores remotos como se fossem chamadas de procedimento local;

- DCE (Distributed Computing Environment). Oferece um ambiente de comunicação que permite as informações fluírem do lugar onde está armazenada para o lugar onde está sendo requisitada, sem expor as complexidades da rede para o usuário final, para o administrador do sistema ou para o desenvolvedor de aplicação. Sua arquitetura mascara as complexidades físicas do ambiente, oferecendo uma camada de simplicidade lógica, composta de um conjunto de serviços que podem ser usados separadamente, ou em combinação;

- ANSA (Advanced Network Systems Architecture). É uma implementação do modelo de referência ODP (Open Distributed Processing). Tem como objetivo principal permitir a interoperabilidade entre aplicações e também o compartilhamento de informações entre instituições. Consiste em um conjunto de níveis de abstração que descrevem a estrutura do sistema distribuído sob cinco pontos de vista: empresa, informação, computacional, engenharia e tecnologia. Esta arquitetura permite o desenvolvimento de diferentes projetos, cada qual direcionado para uma aplicação particular ou uma área de aplicação;

- Java/RMI. Java é uma linguagem de programação caracterizada por sua portabilidade, robustez, segurança, suporte a programação distribuída e amplos recursos para o desenvolvimento de aplicações multimídia. A quase perfeita portabilidade das aplicações desenvolvidas em Java é uma das suas grandes vantagens em sistemas heterogêneos. Java RMI é similar ao mecanismo de RPC (Remote Procedure Call), só que escrita totalmente em Java. A limitação do RMI é exatamente o seu uso em ambientes heterogêneos, a exemplo da interação com objetos desenvolvidos em outras linguagens.

- DCOM (Distributed Component Object Model). É um meio de comunicação entre componentes remotos ActiveX e outros serviços. Neste sentido, o DCOM tornou-se um ORB para o ActiveX. É baseado na filosofia de orientação a objeto e utiliza o conceito de interface, através de IDL; a chamada do cliente para o servidor, utilizando as invocações estáticas e dinâmicas; um repositório de interfaces destinado a invocar e localizar objeto semelhante ao CORBA, chamado de Type Library. É uma tecnologia desenvolvida pela Microsoft e que somente está disponível para os sistemas operacionais Windows NT e Windows 95. Isto significa dizer que a idéia de heterogeneidade ficará prejudicada assim como a interoperabilidade entre máquinas somente existirá quando utilizado um dos sistemas operacionais acima mencionados.

Como para esta dissertação adotamos a tecnologia CORBA, pelo seu suporte a sistemas heterogêneos, faremos em seguida uma melhor explanação sobre este modelo.

2.3.1. CORBA

Talvez a proposta mais promissora para a computação distribuída de objetos seja a da OMG – Object Management Group - um consórcio apoiado em mais de 800 empresas membros. A OMG é uma organização sem fins lucrativos fundada em 1989, que se dedica a divulgação dos padrões de orientação a objetos. A OMG tem definido um conjunto de especificações - chamado Object Management Archictecture (OMA) - para a computação distribuída. Em dezembro de 1990 foi publicado um documento pela OMG, denominado Object Management Architecture Guide. A plataforma para as aplicações é o Common Object Request Broker Architecture – CORBA e seus serviços adicionais [BETZ95].

Embora as tecnologias apresentadas anteriormente possam ser utilizadas no gerenciamento distribuído, optou-se aqui pela utilização do CORBA, pelas seguintes razões:

- CORBA combina os conceitos de reutilização de software, através das técnicas de orientação a objetos, aos conceitos de computação distribuída;
- CORBA permite que as aplicações acessem e compartilhem os objetos entre si, tornando estes comuns a todas as aplicações que implementem CORBA;
- CORBA está se tornando uma solução para a interoperabilidade entre objetos que não ficam presos a uma plataforma ou padrão específico.

2.3.1.1. CORBA e IIOP

Para que o CORBA, garantisse a interoperabilidade entre objetos pertencentes a diferentes implementações de ORBs, foi necessário a padronização do protocolo de comunicação e do formato de mensagens. CORBA 2.0 apresenta estas especificações definidos nos seguintes protocolos:

- Protocolo Inter-ORB Geral (GIOP): especifica um conjunto de formatos das mensagens e dados para a comunicação entre ORBs;
- Protocolo Inter-ORB Internet (IIOP): especifica como as mensagens GIOP são transmitidas numa rede TCP/IP (GIOP + TCP/IP = IIOP);
- Protocolo Inter-ORB para Ambiente Específico (ESIOPs): é uma especificação para permitir a interoperabilidade do ORB com outros ambientes (ex.: DCE, DCOM).

2.3.1.2. Arquitetura CORBA – OMA

A arquitetura CORBA foi publicada em 1990 no Object Management Group Guide como sendo orientada a objetos. A OMG completou seu objetivo com o estabelecimento de uma arquitetura chamada Object Management Arquitecture (OMA) – Arquitetura de Gerenciamento de Objetos - que é a base do CORBA. As quatro partes que fazem parte do OMA estão apresentadas na figura 5.

Em CORBA o sistema intermediário, ou middleware, que realiza o mapeamento dos requisitos de serviços do cliente para uma implementação de servidor é chamado de Broker – Basic Object Request Broker. A especificação CORBA define a arquitetura de um ORB, cujo papel é habilitar e regular a interoperabilidade entre objetos e aplicações [BETZ95]. O ORB é o componente a partir do qual o CORBA derivou o seu nome, pois é

o centro de comunicação para todos os objetos do sistema distribuído e foi o primeiro componente especificado pela OMG a ser implementado.

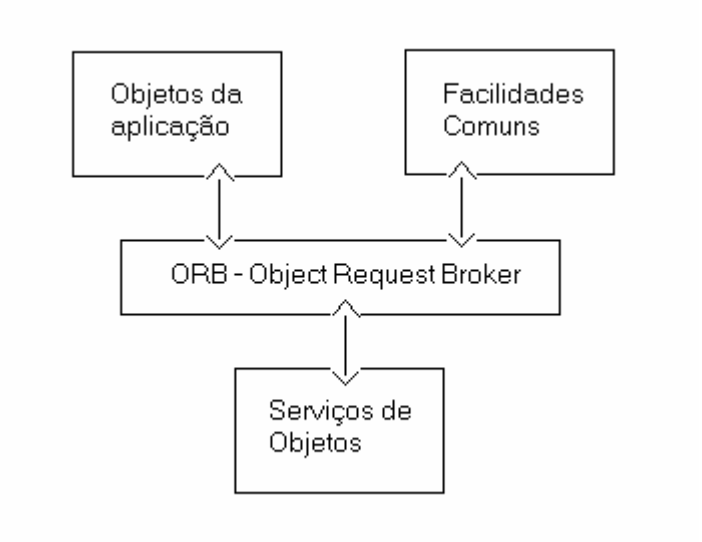

Figura 5 – OMA – Object Management Architecture.

O Broker ou ORB possibilita um fraco acoplamento entre clientes e servidores, gerenciando a interação entre objetos. Este gerenciamento inclui todas as responsabilidades de um sistema distribuído, desde a localização e referenciação dos objetos, até as ações de marshaling dos parâmetros requisitados e resultados. Assim sendo, apenas ele precisa saber a localização e as capacidades dos clientes e servidores CORBA na rede, sendo que múltiplos servidores podem trabalhar com um único cliente e um único servidor pode operar com múltiplos clientes.

O ORB utiliza informações da requisição para determinar a melhor implementação que pode satisfazê-la. Esta informação inclui a operação que o cliente está requisitando, o tipo de objeto sobre o qual a operação está sendo executada e qualquer informação adicional armazenada em objetos de contexto para a requisição.

A figura 6 mostra como uma aplicação do cliente invoca um método de uma implementação X.1 alocado no servidor X. Tais implementações são obviamente sistemas compostos por objetos possuidores de interfaces que possibilitam a comunicação

através do ORB. As definições destas interfaces serão abordadas mais adiante no final deste capítulo. Note que uma outra implementação X.1 está disponível em um outro servidor Y. Neste caso, o ORB procura realizar o balanceamento de carga dos servidores, ou seja, intercalar chamadas consecutivas entre os servidores X e Y.

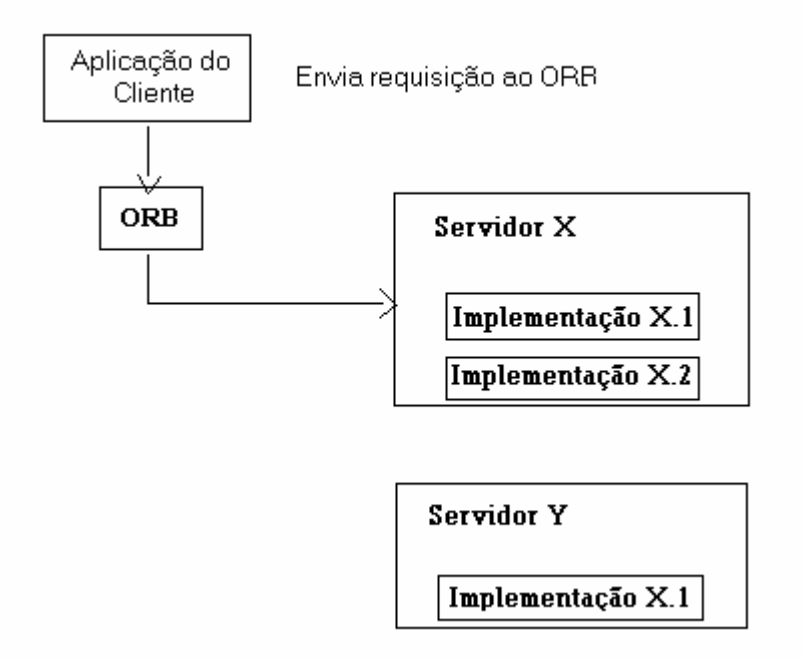

Figura 6 – Seleção de implementação pelo ORB.

Além de selecionar a implementação para realização da requisição, o ORB também valida cada requisição e seus argumentos e pode fornecer informação de autenticação e autorização. Por exemplo, o ORB verifica se o objeto especificado na requisição é valido para a operação especificada. O ORB também procura no repositório de implementação um objeto que possa servir a requisição do cliente em invocações dinâmicas.

Apesar de ser um componente singular, possui ele algumas funções específicas para o lado do cliente e outras funções específicas para o lado do servidor. No lado do cliente, o ORB manipula pedidos de invocação e a seleção de servidores e métodos. Quando uma aplicação envia um pedido para este, visando uma operação em um objeto,

o ORB valida os argumentos a partir da interface e despacha a requisição para o servidor, inicializando-o quando necessário. O ORB é ligado à aplicação e constitui uma parte do cliente.

A partir do CORBA versão 2.0 é possível para tais sistemas possuírem diversos ORBs. Múltiplos ORBs podem possuir diferentes representações para referências de objetos e diferentes estilos de invocação. Se uma aplicação do cliente possuir acesso a duas referências de objetos gerenciados por ORBs diferentes, estes devem distinguir as referências dos objetos [VISI97].

As facilidades comuns não fazem parte da especificação CORBA, mas de uma parte complementar ao OMA, adicionando funcionalidades. As facilidades diferem dos serviços CORBA por especificarem funcionalidades em níveis de abstração mais elevados. As comuns fornecem um conjunto de aplicações de uso geral para diferentes aplicações, como o gerenciamento de documentos, o acesso a base de dados, impressão de arquivos ou sincronização de tempo em um ambiente distribuído.

Os serviços CORBA também não fazem parte do CORBA, são uma parte complementar da arquitetura OMA que o OMG define. Os serviços de objetos fornecem um conjunto de serviços em tempo de execução que auxilia na criação de aplicações robustas orientadas à objetos. O uso dos serviços de objetos pode facilitar a criação das aplicações pois permitem ao desenvolvedor chamar funções previamente implementadas ao invés de escrever e chamar funções particulares de serviços. Todas as empresas que desejarem implementar serviços de objetos, devem estar de acordo com as especificações da OMG [GEMS97]. Isso permite a troca de fornecedor de ORBs sem ter que alterar a implementação, desde que não sejam utilizados recursos proprietários.

Os serviços estão especificados no COSS [OBJE97], ou Common Ojbect Services Specification. São definidos alguns serviços como: serviço de nomes, para localização de objetos; serviço de eventos, para transmitir eventos entre objetos; serviço de ciclo de vida, para controlar a criação dos objetos; serviço de persistência, para armazenar os objetos, etc.

2.3.1.3 Componentes e Interfaces da Arquitetura CORBA

Em muitos sistemas distribuídos os clientes e os servidores são fortemente acoplados, sendo que cada parte possui muitas informações sobre a tarefa do outro. Inicialmente, isso facilita a implementação, porém, a necessidade de alterações futuras exigirá a modificação das partes.

Na arquitetura CORBA, o cliente e o servidor estão formalmente separados, sendo que o cliente CORBA sabe apenas como invocar um método e o servidor CORBA sabe apenas como cumprir a tarefa. Isso significa que pode-se modificar a forma como o servidor executa a tarefa sem afetar a forma como o cliente pede ao servidor para realizar a tarefa. Portanto, essa característica de CORBA promove a flexibilidade tão almejada para a implementação de sistemas.

A figura 7 procura resumir a arquitetura de um sistema CORBA, mostrando a interconexão dos seus componentes [OBJE97]. Embora a figura 7 ilustre sistemas de computadores que separam o cliente e o servidor, todos os componentes poderiam estar em uma mesma máquina.

2.3.1.4. Modelo de comunicação CORBA

A combinação da computação distribuída e o modelo de objetos em CORBA melhora tanto a computação distribuída quanto a computação orientada a objetos. A incorporação do modelo de objetos em CORBA fornece uma forma de abstrair as complexidades inerentes do ambiente cliente/servidor. A computação orientada a objetos complementa os benefícios da computação distribuída porque esta esconde as funções de comunicação entre os objetos, como a descoberta da disponibilidade, localização e os

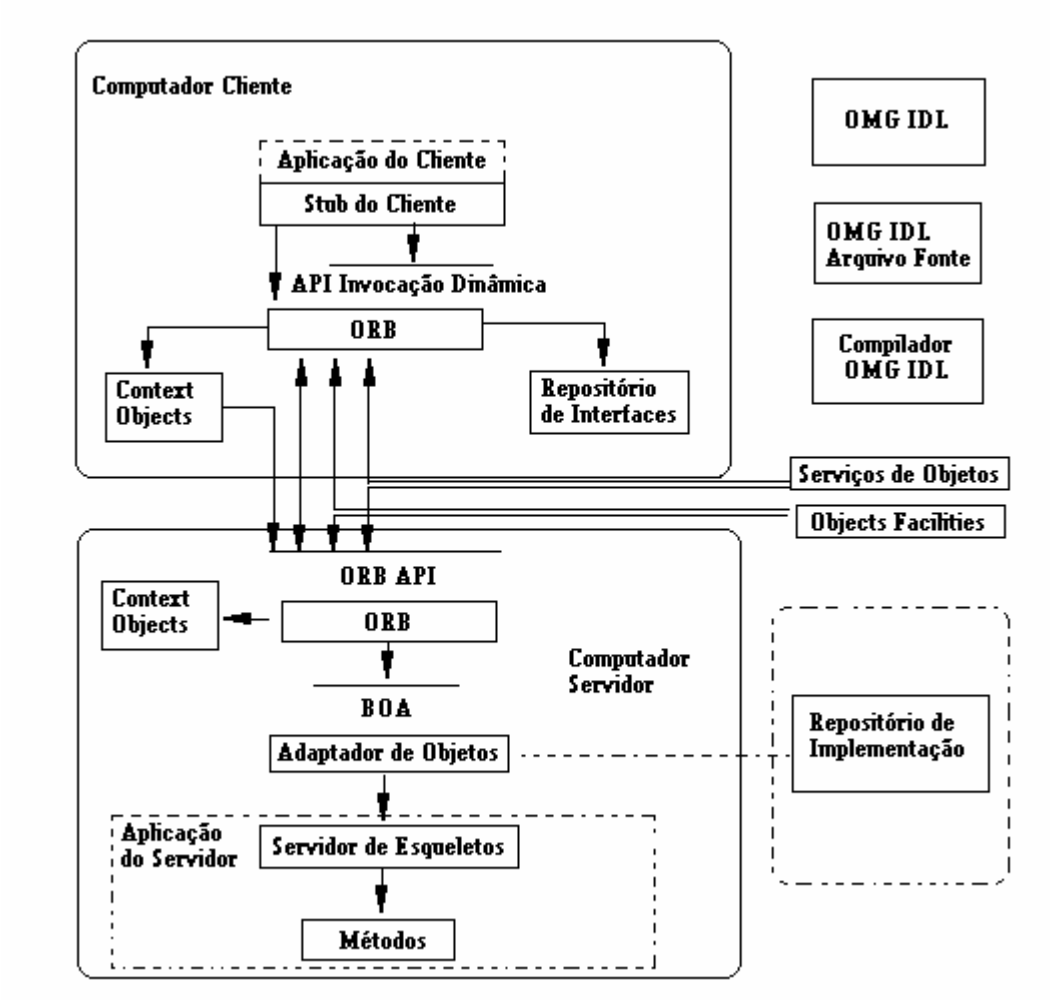

Figura 7 – Componentes e interfaces CORBA.

mecanismos de acesso ao servidor. A figura 8 apresenta o modelo de comunicação CORBA, onde o cliente invoca métodos utilizando o stub do cliente e o skeleton do servidor em uma implementação de objeto. CORBA adiciona referências particulares para objetos no ambiente computacional. Através destas referências os objetos são identificados no sistema.

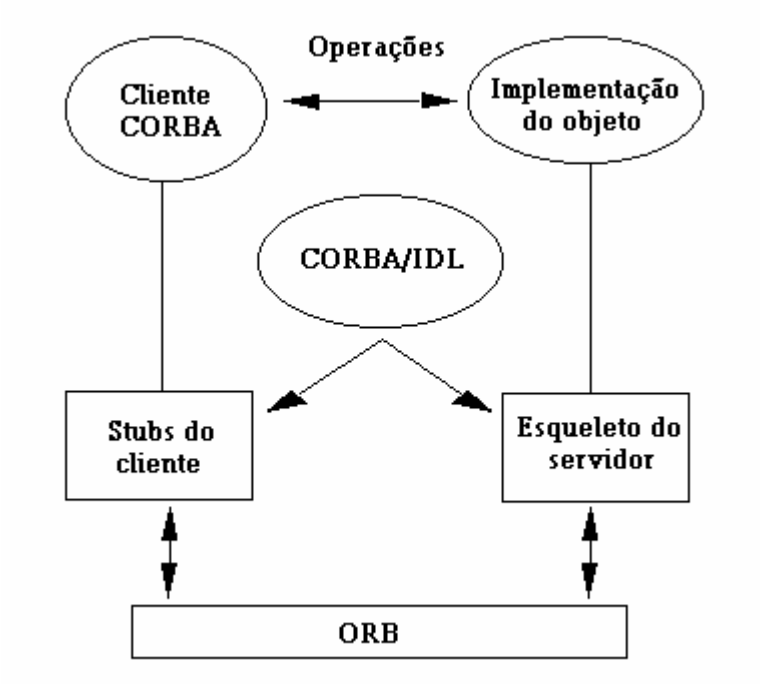

Figura 8 – Modelo de comunicação CORBA.

CORBA suporta tanto o estilo de comunicação síncrona quanto a assíncrona. A comunicação assíncrona é realizada pelo cliente perguntando se a operação foi completada. CORBA também define requisições to tipo one-way, onde a aplicação não necessita esperar que a requisição seja completada e nenhum argumento é retornado.

Esta capacidade do CORBA de suportar diversos estilos de comunicação possibilita que aplicações de gerenciamento de infra-estrutura tornem-se de fácil implementação, pois os tratamentos de concorrência, pooling e envio de mensagens são transparentes aos implementadores.

2.3.1.5. Object Management Group (OMG) IDL

As interfaces para os servidores podem ser especificadas em uma linguagem simbólica e abstrata. A representação de interfaces nestes moldes é anterior à especificação CORBA e aos sistemas orientados a objetos. Este conceito é utilizado em implementações do tipo RPC – Remote Procedure Call.

O propósito de um arquivo IDL é permitir a definição de interfaces independentes de linguagem. Isto é obtido pelo mapeamento entre a sintaxe IDL – Interface Definition Language - e qualquer linguagem que utiliza para implementar os clientes e servidores. Assim sendo, os clientes e servidores não necessitam ser implementados utilizando a mesma linguagem.

Os desenvolvedores de aplicação CORBA utilizam o OMG IDL para definir a estrutura de dados e as características de interface para objetos. Cada operação definida em uma interface consiste no nome da operação, argumentos, tipos de dados, e retorno de valores. O OMG IDL não é uma linguagem de programação e não pode ser utilizado para gerar objetos ou códigos executáveis.

A linguagem CORBA IDL obedece às mesmas regras léxicas que C++ e inclui várias palavras-chave específicas para o uso em sistemas distribuídos.

Embora não seja requisitado na especificação CORBA, o compilador OMG IDL é uma das partes mais importantes deste sistema. Ele gera o stub do Cliente para a invocação do tipo stub e gera o skeleton do servidor. Um compilador OMG IDL está incluso na maioria dos sistemas CORBA comercializados.

Pode-se utilizar o arquivo OMG IDL para definir tipos de objetos pelas características de suas interfaces. O corpo de uma interface declara as constantes exportadas, declaração de tipos, exceções, atributos, e operações. Definições de operações incluem nome da operação, tipo de dado retornado, tipo de todos os argumentos, exceções retornáveis e informações contextuais que poderiam afetar os dados despachados.

2.4. Orientação a Objeto

O conceito fundamental de Objeto é o de uma entidade, cujas informações podem incluir desde suas características, ou dados (Atributos), até os procedimentos para a manipulação dessas características (serviços). A um objeto sempre estarão associados seu estado, comportamento e identidade. O estado é definido pelas propriedades do objeto e pelos valores que as propriedades possuem. O comportamento define como o objeto age e reage, tanto em relação à mudança em seu estado, como à comunicação com outros objetos. A identidade de um objeto é a propriedade pela qual ele se distingue dos demais [KHOS86].

Quando os objetos possuem características comuns, pode-se agrupá-los em uma classe, na qual são definidos os dados e procedimentos associados a todos os objetos da classe. Este tipo de entidade denomina-se Classe & Objetos. Algumas classes são generalizações e não possuem objetos. Estas são denominadas Classes. Dados e procedimentos associados a uma Classe ou a uma Classe & Objeto são chamados de Membros da classe.

O primeiro tipo de membros de uma classe são os Atributos. Eles descrevem as propriedades do objeto (estado). A manipulação dos atributos é efetivada pelos membros denominados Serviços (ou métodos) associados à classe.

Quando o atributo deve ser distinto para cada objeto, ele caracteriza um Atributo de Objeto. Este pode ter, até mesmo idênticos valores, mas é entidade separada no sistema. Além deste, uma classe pode possuir atributos cujos valores devem ser compartilhados entre todos os seus objetos. A este, dá-se o nome de Atributos de Classe. Quando um Objeto modifica o valor de um Atributo de Classe, todos os demais objetos da classe têm seus valores modificados.

Um exemplo de objeto pode ser obtido quando consideramos um programa processador de textos. Nele podemos definir a seguinte Classe & Objeto:

Classe & Objeto : parágrafo;

Atributos : fonte e textos do parágrafo;

Método : formata-se.

Um objeto desta classe seria, por exemplo, o parágrafo que se está lendo.

Ao definir uma Classe, é necessário que se especifique o tipo de acesso que seus membros (atributos e serviços) terão: o acesso público permite que todas as outras entidades do sistema (funções e demais classes) acessem este membro; o acesso protegido permite que as classes, na hierarquia da classe em questão possam acessar os membros sob essa especificação; finalmente, temos os membros que só devem ser acessados pelos serviços da classe, possuem acesso privado.

Quando um programa orientado a objeto está sendo executado, ocorre um processo de comunicação entre objetos, através do envio de mensagens (chamadas de funções ou procedimentos). Estes procedimentos definem o comportamento que o objeto receptor deverá ter. Para alcançar este objetivo, o objeto aciona aqueles serviços relacionados à mensagem enviada. Todo o processamento é definido pelo envio, pela interpretação e pelas respostas às mensagens entre objetos.

À medida que as características de objetos semelhantes vão sendo agrupadas, forma-se uma Classe. Isto redefine o Objeto como sendo sempre a Instância de uma classe.

Outro mecanismo importante na Orientação a Objetos, é o mecanismo de agrupamento de características, em que algumas classes (filhos) são variações especializadas de outras (pais), constituindo a Herança. Como exemplo, pode-se citar o fato de que alguns atributos da classe (automóvel modelo, ano, etc.) também estão presentes em uma classe mais geral, denominada veículo, cujas subclasses, além de automóvel, comportam avião, navio, etc.

Quando a classe derivada possui características em mais de uma classe-base, a hierarquia constitui uma Herança Múltipla.

A Abstração consiste na formulação do problema sob a ótica mais adequada à solução. Na análise orientada a objeto, todo o processo de definição de Classe, Hierarquia, Atributos, Métodos e Mensagens, que formarão o programa orientado a

objeto, é fruto da abstração do analista e, como tal, é de suma importância para o alcance dos objetivos.

Na programação fundamentada em procedimentos os programas são baseados em coleções de funções. Estas modelam operações abstratas cujo intuito é resolver um problema de programação. Na Abstração de Dados, o foco está nas estruturas de dados e não nas operações que se fazem com eles. Elas são, inclusive, parte do dado que se modela, em outras palavras, a abstração no paradigma da Programação Estruturada está nos procedimentos. No paradigma da Orientação a Objeto, tem-se dados Abstratos, onde uma estrutura de dados inclui as operações que ocorrem com ela. Para efetivar este conceito, a técnica usada é encapsular dados e procedimentos em um tipo abstrato de dado.

Uma das principais diferenças entre a Análise Estruturada e a Análise Orientada a Objetos, é a característica do encapsulamento dos objetos. Segundo ela, a definição de um objeto inclui tanto os seus atributos como os métodos que agem sobre esses atributos. Na Análise Orientada a Objetos, não se separam dados de procedimentos, como se faz na Análise Estruturada.

Outra característica relevante da Orientação a Objetos é o Polimorfismo. Trata-se da propriedade de métodos com diferentes códigos e de diferentes níveis na hierarquia de classes que, apesar de possuírem o mesmo nome, podem ser diferenciados pelo contexto em que estão sendo chamados. Exemplo comum é a operação de nome *on* (ligar), em vários eletrodomésticos, computadores, máquinas sofisticadas, etc. Embora os nomes sejam idênticos, os procedimentos em cada situação são, evidentemente, diferentes sendo o contexto que define a quais desses procedimentos está-se referindo.

Um sistema orientado a objetos deve apresentar, também, modularidade. As abstrações semelhantes devem ser agrupadas em módulos independentes que, quando juntos, formam o sistema. Isto permite maior flexibilidade e eficiência na implementação do sistema.

Outro conceito-chave da Orientação a Objetos é Persistência. Objetos persistentes são aqueles que permanecem existindo, mesmo após o término da execução do programa.

Associados à persistência, estão o gerenciamento dinâmico da memória e o armazenamento de objetos em base de dados. Os objetos não-persistentes devem ter seu espaço de memória dinamicamente gerenciados, ou seja, alocados quando necessário e liberados para outro uso, quando não mais importar no processamento do programa. Em termos de memória auxiliar, os objetos armazenados em bases de dados orientadas a objetos são ditos persistentes.

Uma característica desejável no modelo da Orientação a Objetos é a Tipificação.

O uso do objeto de uma classe, onde se faz necessário a utilização de objeto de outra classe, só deve ser permitido sob condições controladas (conversões explícitas). Dentre as linguagens de Orientação a Objetos, encontraremos linguagens fortemente ou fracamente tipadas ou mesmo linguagens não tipadas, dependendo da liberdade que a linguagem fornece no intercâmbio de tipos.

Várias são as notações disponíveis para Sistemas Orientados a Objetos. As principais são: a de Booch ([BOOC92]; [BOOC94]) e a de Coad e Yourdon [COAD92]. Para estas, inclusive, encontramos ferramentas (CASE) no mercado para a especificação de Sistemas Orientados a Objetos. Ambas metodologias são relevantes: Booch, por sua contribuição ao nível de definição e ilustração de conceitos como Coad; Yourdon, por apresentar a notação gráfica, que nos parece mais clara.

2.5. Linguagem Java

Java é uma linguagem de programação com características muito similares ao C++, cuja utilização cresce principalmente em desenvolvimento para internet através de Applets, Servelets e JSP – JavaServer Pages, que por serem escritos em Java tem a disposição todas as APIs e softwares disponíveis na plataforma, o que lhes conferem alta flexibilidade, representada por uma linguagem orientada a objetos, multiplataforma, multithreading e com forma de acesso a bases de dados padronizada. Logo a linguagem Java apresenta características bem peculiares que justificam e viabilizaram a implementação deste projeto.

Ao ser compilado um programa em Java, é gerado um código para uma máquina genérica denominado bytecode. Este código pode ser executado tanto por interpretadores, como por Browsers Web (que simulam a implementação de uma máquina virtual baseada em bytecode). Esta característica confere aos programas construídos em Java a peculiaridade de serem independentes de plataforma.

O JavaServer Pages - JSP é uma tecnologia baseada na linguagem Java, que simplifica o processo de desenvolvimento dinâmico de sites Web. O JSP funciona como um compartimento (container) que incorpora elementos dinâmicos.

O JSP funciona no lado do servidor. Ou seja, as páginas JavaServer são arquivos texto, normalmente com a extensão ".jsp" que substituem os arquivos estáticos HTML. As páginas JSP, além de utilizar objetos do servidor, podem incorporar e manipular objetos próprios, como Applets e Servlets.

A linguagem Java tinha o propósito de gerar pequenos programas locais (stand-alone) nomeados Applets, incrementando assim funcionalidades ao browser do cliente. Entretanto em 1996, para adicionar dinamismo ao lado do servidor, criou-se os Servlets. E o JSP veio para integrar todas essas tecnologias.

O JSP é alguma coisa como um sistema híbrido de templates, ora parecido com o ColdFusion, ora como ASP, SSJS e PHP. A linguagem padrão utilizada nas JSP's é Java puro. Mas qualquer tipo de linguagem script pode ser utilizada no lugar do Java, até XML e ColdFusion. As novas especificações do JSP implementam tags especiais que substituem o código Java puro dentro da página JSP.

3. PADRÕES DE PROJETO

Nos últimos anos, verificou-se o grande aumento da utilização da tecnologia de orientação a objetos. Desde o início da década de 90, tem-se acompanhado, a princípio na comunidade acadêmica e posteriormente nas empresas, a crescente adoção da Orientação a Objetos para o desenvolvimento de aplicativos. Este aumento deve-se, principalmente, às soluções propostas por esta tecnologia, as quais vêm amenizar os efeitos dos problemas gerados pela Crise de Software. No começo, algumas empresas viam com ressalvas os benefícios trazidos pela orientação a objetos, devido à baixa produtividade inicial conseguida por seus desenvolvedores. Entretanto, passados alguns anos, não existem mais dúvidas quanto a sua adoção, sendo apenas uma questão de tempo a sua completa utilização, e também de seus princípios.

Como conseqüência da adoção da metodologia orientada a objetos para o desenvolvimento de aplicativos, tivemos uma explosão de métodos, linguagens e ferramentas que implementam, de uma forma ou de outra, todos os conceitos apresentados por esta tecnologia. Hoje existe uma tentativa de se padronizar uma linguagem para a especificação de sistemas orientados a objetos, a UML. Esta linguagem é baseada em um conjunto de métodos já utilizados e adotados em diversos lugares do mundo.

Embora tendo uma tecnologia tão construtiva, a simples adoção dos conceitos não é o suficiente para a construção de aplicativos com qualidade, mesmo que acompanhados das melhores ferramentas.

Projetar um software orientado a objetos é algo trabalhoso, mas projetar software reutilizável orientado a objetos é ainda uma tarefa muito mais engenhosa [ERIC00]. Devem se achar objetos pertinentes, fatorá-los em classes no nível correto de granularidade, definir as interfaces das classes e as hierarquias de herança e estabelecer as relações-chave entre eles. O projeto deve ser específico para o problema a ser resolvido, mas também genérico o suficiente para atender futuros problemas e requisitos. Deseja-se também evitar o re-projeto ou, pelo menos, minimizá-lo. Os mais experientes projetistas

orientados a objetos nos dirão que um projeto reutilizável e flexível, é impossível de ser obtido na primeira vez e de maneira correta. Antes que um projeto esteja terminado, normalmente tenta-se reutilizá-lo várias vezes, modificando-o a cada utilização [ERIC00].

Isto ocorre porque, normalmente, todos os projetos envolvem a resolução de problemas recorrentes. E quantas vezes não nos pegamos com a sensação de já termos resolvido um problema semelhante em algum momento do passado? Este conhecimento de como se resolver um determinado problema, acumulado durante os anos, é o que chamamos de experiência e esta está ligada diretamente aos responsáveis pela resolução, sendo carregada apenas pelos desenvolvedores que um dia resolveram aquele problema. Mas então, como podemos armazenar esta experiência para outros usos, ou então para auxiliar outros desenvolvedores?

Este ponto é importante: um desenvolvedor experiente nunca começa a criar um sistema realmente do zero. Ele aproveita desenvolvimentos anteriores que deram certo e que foram refinados com o tempo como ponto de partida para um novo sistema.

Se pudéssemos catalogar esta experiência anterior em uma forma que qualquer pessoa, experiente ou não, conseguisse realmente utilizar para resolver seus problemas de desenvolvimento, conseguiríamos um grande aumento na qualidade dos projetos e uma redução sistemática no tempo de desenvolvimento dos sistemas.

Os conceitos de tempo para desenvolvimento de softwares que estão sendo colocados para os programadores por algumas companhias como Microsoft, Netscape etc., estão ajudando ainda mais a consolidar a crise de software existente na indústria.

Foi pensando nisso que a comunidade de software buscou maneiras de se superar a crise. Olhando outras disciplinas, verificou-se que na engenharia civil buscava-se em projetos anteriores sucessos e fracassos dos quais pudessem tirar algum proveito.

Um dos principais motivos levantados para a catalogação da experiência é fazer com que desenvolvedores novatos possam agir como se fossem desenvolvedores especialistas, sem a necessidade de passar anos ganhando experiência.

3.1. Histórico

No final da década de 70, Christopher Alexander passava por um problema semelhante, só que relacionado à arquitetura e à engenharia civil. Percebendo que todas as construções de edifícios, os quais eram funcionais e confortáveis, possuíam algumas características em comum, resolveu catalogar estas soluções em seu livro “The Timeless Way of Building”.

Durante anos, os especialistas na área de engenharia de software tentaram adequar as idéias de Alexander para a construção dos sistemas de informação. Finalmente, em 1994, liderados por Erich Gamma a GoF (Gang of Four) – Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides - publicaram o livro que deu origem a onda dos Padrões na área de informática, os Design Patterns: Elements of Reusable Object-Oriented Software. O primeiro catálogo bem descrito sobre Padrões de Projeto para programas orientados a objeto.

3.2. Definindo Padrões de Projeto

Segundo Christopher Alexander “cada padrão descreve um problema no nosso ambiente e o núcleo da sua solução, de tal forma que se possa usar esta solução mais de um milhão de vezes, sem nunca fazê-lo da mesma maneira” [ALEX97]. Embora Alexander estivesse falando acerca de padrões em construções e cidades, o que ele diz é verdadeiro em relação aos padrões de projetos orientados a objeto. As soluções são expressas em termos de objetos e interfaces para um problema num contexto.

Em [ERIC00], um padrão tem quatro elementos essenciais:

1. O nome do padrão é uma referência que podemos usar para descrever um problema de projeto, suas soluções e conseqüências em uma ou duas palavras;

2. O problema descreve quando aplicar o padrão;

3. A solução descreve os elementos que compõem o projeto, seus relacionamentos, suas responsabilidades e colaborações;

4. As conseqüências são os resultados e análises das vantagens e desvantagens (trades-offs) da aplicação do padrão. Uma vez que a reutilização é freqüentemente um fator no projeto orientado a objetos, as conseqüências de um padrão incluem o seu impacto sobre a flexibilidade, a extensibilidade ou a portabilidade de um sistema.

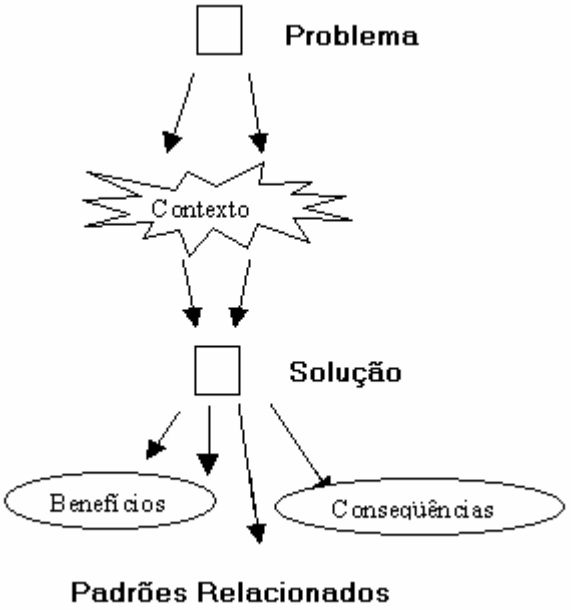

Figura 9 - Mostra a convergência/divergência dos padrões.

Para Thomas Mowbray Padrões de Projeto é um par entre um problema e solução, acompanhado por uma associação a um determinado contexto e uma identificação da força que mais define o problema [MOWB97]. É reutilizável, logo é aplicável a uma gama de problemas de projeto em uma larga variedade de circunstâncias. Padrões de Projeto é expressa em termos de um esboço fixo, freqüentemente chamada de

modelo, onde existem alguns elementos que ajudarão a responder importantes questões sobre o padrão [MOWB97].

3.3. Linguagens de Padrões de Projeto

O pai de Padrões de Projeto é Christopher Alexander e sua meta está em prover a qualidade de padrões arquiteturais, analisando as forças que envolvem um problema arquitetural e reutilizando as soluções padrões que tiveram sucesso para outros projetos com problemas similares. Durante anos, especialistas na construção de sistemas de informação trabalharam em seus modelos, dos quais apresentamos os mais conhecidos.

3.3.1. Linguagem de Padrão Buschmann

Buschmman [BUSC96] propõe um modelo com itens que permitem a melhor visualização da arquitetura inserida na solução proposta. Uma arquitetura é composta por uma parte estática e outra dinâmica. Ele propôs 14 itens para fazer a descrição de padrões, sendo que a utilização destes pode ser opcional.

- Nome: contém o nome do padrão, deve ser intuitivo;
- Exemplo: um exemplo do mundo real demonstrando a existência do problema e a necessidade de um padrão;
- Contexto: as situações em que o padrão pode ser utilizado ou seja, descreve o contexto destas situações;
- Problema: descreve a questão que o padrão resolve;
- Estrutura: descreve o princípio fundamental da solução;
- Dinâmica: descreve o aspecto dinâmico da solução encontrada pelo padrão;
- Implementação: algumas diretrizes para a implementação do padrão;
- Variantes: descrição de variantes ou especializações do padrão;

- Usos conhecidos: contém uma breve descrição dos sistemas existentes que podem se beneficiar do uso do padrão;
- Conseqüências: descreve os benefícios do padrão e suas habilidades potenciais;
- Também conhecido como: outros nomes para padrões, se quaisquer são conhecidos;
- Resolução do exemplo: discussão de qualquer aspecto importante para resolução do exemplo que ainda não estão cobertos nas seções de solução estrutura, dinâmica e implementação;
- Veja também: referências para padrões que resolvem problemas similares e para padrões que ajudam a refinar o padrão que está sendo descrito.

3.3.2. Linguagem de Padrão Gamma

Para Gamma, as notações gráficas, embora sejam importantes e úteis, não são suficientes. Elas simplesmente capturam o produto final do processo de projeto como relacionamentos entre classes e objetos. Para reutilizar o projeto, devemos também registrar decisões, alternativas e análises de custos e benefícios que incidiram sobre o mesmo. Também são importantes exemplos concretos, porque ajudam a ver o projeto em ação.

O formato apresentado por Gamma fornece uma estrutura uniforme às informações, tornando os padrões de projeto mais fáceis de aprender, comparar e usar.

O formato Gamma possui os seguintes itens:

- Nome e classificação do padrão: o nome do padrão expressa a sua própria essência de forma sucinta. Um bom nome é vital, porque ele se tornará parte do seu vocabulário de projeto;
- Intenção e objetivo: é uma curta declaração que responde às seguintes questões: O que faz o padrão de projeto? Quais os seus princípios e sua intenção? Que tópico ou problema particular de projeto ele trata?

- Também conhecido como: outros nomes bem conhecidos para o padrão, se existirem;
- Motivação: um cenário que ilustra um problema de projeto e como as estruturas de classes e objetos no padrão solucionam o problema. O cenário ajudará a compreender as descrições mais abstratas do padrão que vem a seguir;
- Aplicabilidade: quais as situações nas quais o padrão de projeto pode ser aplicado? Que exemplos de maus projetos ele pode tratar? Como se pode reconhecer essas situações?
- Estrutura: uma representação gráfica das classes do padrão usando uma notação baseada na Object Modeling Technique (OMT) [RUMB91]. Nós também usamos diagramas de interação ([JACO92], [BOOC94]) para ilustrar seqüências de solicitações e colaborações entre objetos;
- Participantes: as classes e/ou objetos que participam do padrão de projeto e suas responsabilidades;
- Colaborações: como os participantes colaboram para executar suas responsabilidades;
- Conseqüências: como o padrão suporta a realização de seus objetivos? Quais são os seus custos e benefícios e os resultados da sua utilização? Que aspecto da estrutura do sistema ele permite variar independentemente?
- Implementação: que armadilhas, sugestões ou técnicas precisa-se conhecer quando da implantação do padrão? Existem considerações específicas de linguagem?
- Exemplo de código: fragmentos ou blocos de código que ilustram como se pode implementar o padrão em C++ ou Smalltalk;
- Usos conhecidos. exemplos do padrão encontrados em sistemas reais;
- Padrões relacionados: quais padrões de projeto estão intimamente relacionados com este? Quais são as diferenças importantes? Com quais outros padrões este deveria ser usado?

3.3.3. Linguagem de Padrão Mowbray

Mowbray cria um modelo de escala para mostrar que existem níveis para padrões de projeto, como mostra a figura 10, e que cada nível de escala tem soluções e forças que são únicas. Forças são as preocupações de um projeto dentro de um contexto.

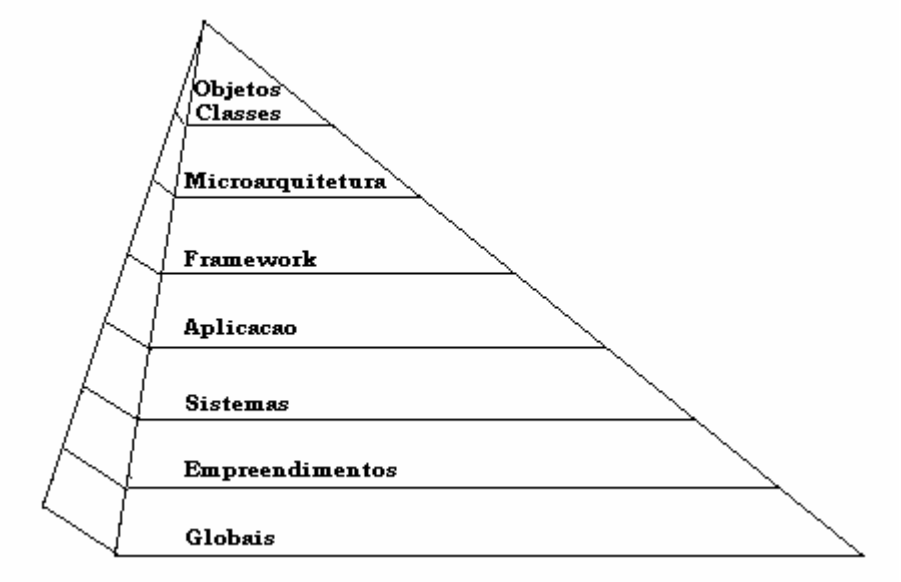

Figura 10 - Mostra os níveis do modelo de escalabilidade.

O formato de Mowbray possui os seguintes elementos:

- Escala: identifica em que modelo de escalabilidade ele se enquadra. Cada padrão é colocado logicamente onde o problema é melhor aplicado;
- Nome: é usado como referência no futuro para conhecimento de princípios contidos em um padrão;
- Intenção: é uma declaração breve do problema;
- Forças primitivas: são identificadas as forças primitivas relacionadas ao modelo de escalabilidade;

- Aplicabilidade: esta seção é uma lista de fatores que motivam o uso deste padrão. Se uma ou mais destes fatores se aplicam, então, este padrão pode ser aplicado ao problema;
- Resumo de solução: define como o problema será resolvido;
- Benefícios: são apresentados os benefícios da solução e realçadas as vantagens em cima de soluções comparáveis;
- Conseqüências: são apresentadas as conseqüências indesejáveis quando da aplicação da solução;
- Exemplos: fragmentos ou blocos de código que ilustram como se pode implementar o padrão em Java, C++ ou Smalltalk.

3.4. Algumas Considerações

O ponto de vista de cada indivíduo afeta o que vem a ser ou não um padrão. Não existe uma definição clara sobre em que ponto da abstração deve-se chegar para a definição de um padrão. A tabela 1 mostra uma relação entre os níveis e os seus construtores.

A maior parte dos autores concorda sobre as principais características dos padrões. Muito se pode aprender sobre eles, como funcionam e suas vantagens através de suas características:

- Endereçam um problema recorrente em uma situação especifica e apresentam uma solução para isto [ALEX97];
- Não são inventados ou criados artificialmente [BUSC96]. São simplesmente soluções que já foram utilizadas em diversos projetos anteriores e que provavelmente serão utilizados no futuro. Tipicamente, padrões descrevem um conjunto de componentes, classes ou objetos e suas responsabilidades e relacionamentos;
- Nomes de padrões, se escolhidos cuidadosamente, tornam-se parte da linguagem de desenvolvedores [BUSC96]. Eles facilitam a discussão dos

problemas e suas soluções, removendo a necessidade de explicação para a resolução de um problema particular, desde que todos conheçam a solução daquele padrão e como ele funciona;

- Identificam e especificam abstrações um nível acima de simples classes e instâncias, ou de componentes [BUSC96]. Normalmente, padrões descrevem um conjunto de componentes, classes ou objetos e detalhes de suas responsabilidades e relacionamentos, bem como suas cooperações;

Níveis\Autores	Mowbray	Buschmann	Gamma	Taligent
Objetos e Classes		X		
Microarquitetura			X	
Frameworks				X
Aplicação	X			
Sistemas	X			
Enterprise	X			
Global	X			

Tabela 1 - Mostra o relacionamento entre os níveis de escala e seus construtores.

- Provêem vocabulário e entendimento comum para os princípios de projeto [BUSC96]. Uma vez que um padrão abstrai uma solução, não existe a necessidade de uma longa e detalhada explicação do problema e o que foi feito para sua solução. Ao citar o nome do padrão utilizado, automaticamente será conhecido o problema, em que contexto se acha e a solução adotada para sua solução. E, ao longo do tempo, estes nomes vão se tornando parte do vocabulário comum de uma equipe de desenvolvimento;
- São um meio para documentação de arquiteturas de software [ALEX97]. Através deles pode-se descrever o que se tinha em mente no momento em que

se estava projetando um aplicativo. Assim, qualquer um que um dia necessite alterar as estruturas deste sistema, terá conhecimento, não só de como está estruturado, mas também do que o levou a ser feito desta maneira e, então, prosseguir com o desenvolvimento seguindo tais premissas;

- Auxiliam na construção de arquiteturas de software complexos e heterogêneos [ALEX97]. Como cada padrão utiliza um conjunto fechado de classes, regras e relacionamentos, torna-se extremamente fácil juntar diversos padrões, os quais se relacionam e colaboram para chegar a um objetivo maior, ou seja, padrões como blocos de montar, através dos quais pode-se construir projetos mais complexos [BUSC96];
- Auxiliam no gerenciamento da complexidade do software [ALEX97]. Quando se está desenvolvendo um novo projeto e se faz uso dos padrões, as classes, relacionamentos e detalhes ficam escondidos. O que se vê é apenas uma abstração da solução e, como tudo, deve funcionar.

4. CATÁLOGO DE PADRÕES DE PROJETO

Um catálogo de padrões é uma coleção dos mesmos. Cada padrão é uma solução para um problema relacionado a um determinado contexto, onde é essencial uma boa definição do problema e uma solução concreta. É um jogo logicamente organizado onde os mesmos podem ser interrelacionados, formando uma linguagem de padrões ou, até mesmo, sendo usados independentemente para alcançar soluções, conforme mostra a figura 11.

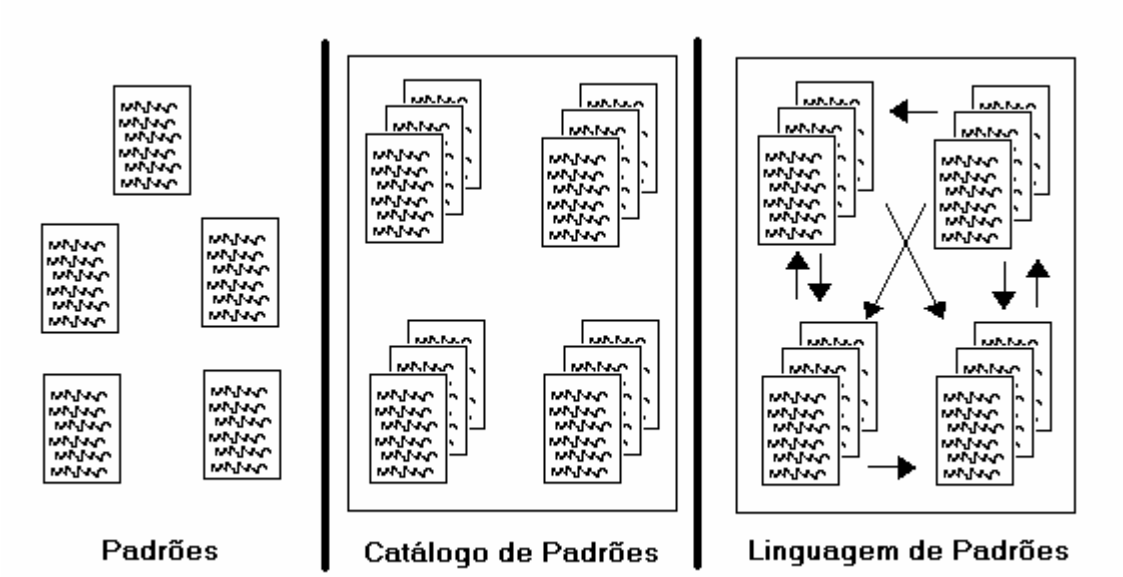

Figura 11. Formas de utilização de padrões.

Este catálogo não tem como propósito fazer uma classificação por critérios de criação, estrutural ou comportamental conforme descreve Gamma, mas sim, visando a utilização de padrões para aplicações distribuídas.

Utilizaremos dois níveis escritos no modelo de escalabilidade descrito por Mowbray: o nível microarquitetura e o nível global.

4.1. Nível Microarquitetura

O nível microarquitetura envolve padrões que combinam múltiplos objetos ou classes. Ele trabalha no desenvolvimento de pequenos projetos, limitando os problemas com software de aplicação. A característica deste nível é um grupo limitado de objetos que cooperam e se inter-relacionam com outros objetos bem definidos e entendidos para a implementação de um componente. A meta de padrões microarquitetural é a reusabilidade de suas estruturas e uma maior facilidade no que tange a extensibilidade para controlar, futuramente mudanças em um sistema. A Linguagem de Padrões Gamma está principalmente ligada ao desenvolvimento efetivo de padrões de projetos neste nível de aplicação, só que não voltado para objetos distribuídos. A seguir temos a descrição dos padrões deste nível.

4.1.1. Padrão Factory

- **Nome:** Factory.
- **Intenção:** em um ambiente distribuído, nem sempre é possível alocar memória para um objeto em um sistema estrangeiro. O padrão factory facilita instanciar objetos remotos.
- **Motivação:** dentro do padrão factory, um único objeto se torna um dedicado objeto factory. Desde então, inúmeros clientes simultâneos poderão fazer uso do padrão factory escrito em thread-safe (linha de execução segura). Se o uso for extremamente alto, ou se uma instanciação de um objeto designado demorar muito tempo, o objeto factory irá responder cada pedido em uma linha de execução provavelmente separada.
- **Aplicabilidade:** este padrão é aplicado em ambiente distribuído, onde instanciar objetos remotos é necessário. Em um ambiente objeto distribuído, sempre será

necessário um cliente instanciar um objeto-servidor. Este padrão resolve tal necessidade conforme figura 12.

- **Estrutura**:

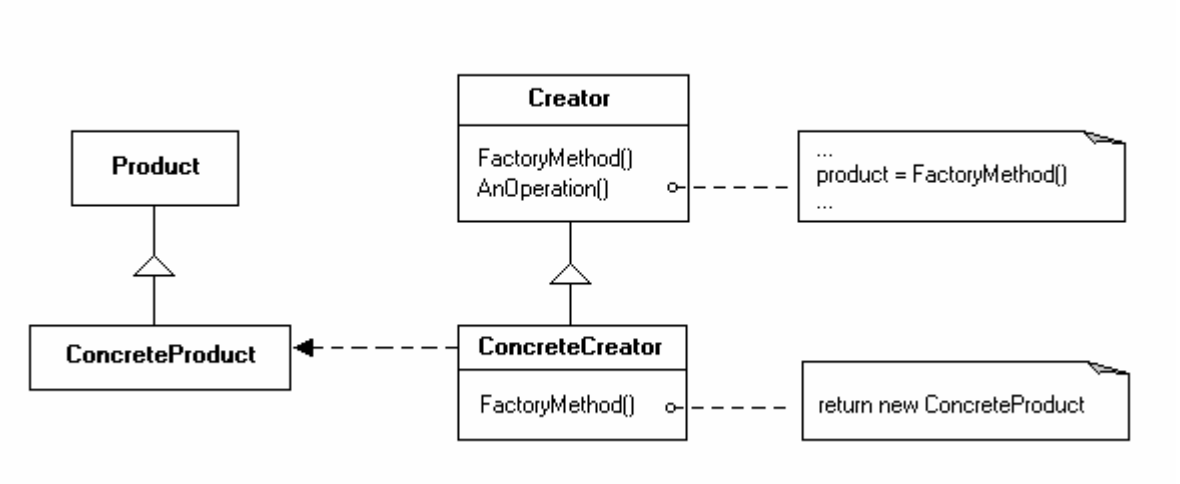

Figura 12 - Estrutura padrão factory.

- **Problema:** quando se executa um programa orientado a objeto distribuído, freqüentemente é necessário instanciar um objeto em uma máquina estrangeira. Enquanto a linguagem de programação Java provê uma palavra-chave para instanciação de um objeto local, não há nenhuma palavra chave explícita que permita instanciar um objeto remoto. Não é possível instanciar um objeto remoto sem utilizar-se da palavra-chave, a menos que o ambiente distribuído faça uso de pedidos de corretores para localizar um objeto em uma máquina remota.

- **Solução:** debaixo do padrão factory, objetos clientes não fazem instanciação explicita para um objeto. Contudo eles ligam para um objeto remoto e pedem que o mesmo execute a instanciação obtendo novos objetos remotos. Este padrão espelha muito a relação produtor consumidor no mundo real. Por exemplo, se você necessita fazer uma camisa, você vai ao alfaiate que tenha suas medidas, escolhe o tecido e leva para o alfaiate costurá-la. Agindo como o padrão factory, o alfaiate aceita o cargo de assegurar que a camisa satisfaça as suas exigências e as do tecido . O objeto-cliente passa dados que agem como exigências para o objeto remoto. O objeto-

servidor usa aqueles dados para instanciar o objeto remoto e retorná-lo ao cliente. CORBA assegura o conhecimento explícito do ambiente remoto que pode assegurar que o objeto instanciado satisfaça o cliente e as exigências do ambiente. A figura 13 mostra o padrão factory em ação.

- **Exemplo**. Três classes são empregadas no exemplo a seguir. Uma classe FactoryServer pode instanciar a classe ServerObject. Uma classe ClientObject primeiro se liga à classe FactoryServer e então instancia o ServerObject.

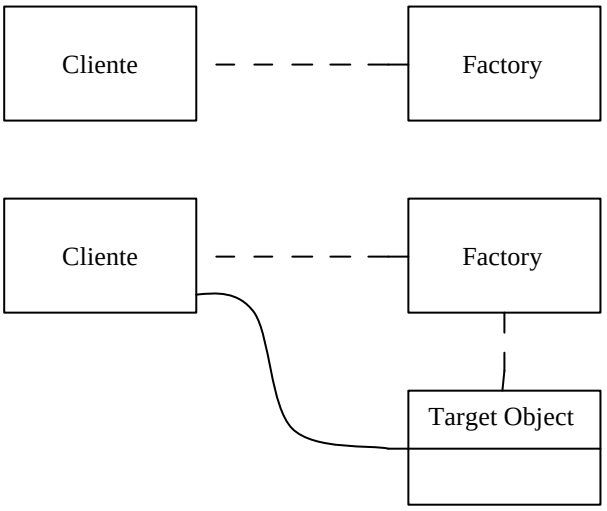

Figura 13 - Padrão factory facilita instanciação de um objeto remoto.

Classe ClientObject

```
package factory;
public final class ClientObject {

 public ClientObject(String[] args) {
   // O código abaixo é específico para tecnologia objetos distribuídos
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
   FactoryServer server = FactoryServerHelper.bind(orb, "FactoryServer");
    // Pede para objeto de fábrica instanciar o ServerObject
   ServerObject serverObject = server.createServerObject("Servidor");
```

```
    System.out.println(serverObject.getName() + " foi criado com sucesso.");
  }
  public static void main(String[] args) {
     ClientObject client = new ClientObject(args);
  }
}
```

Classe FactoryServer

```
package factory;
public final class FactoryServerImpl extends _FactoryServerImplBase {
  public FactoryServerImpl(String name, String[] args) {
     super(name);
     waitForConnection(args);
  }
  private void waitForConnection(String[] args) {
     // Código que especifica a tecnologia objetos distribuídos para habilitação de comunicação
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
    boa.obj_is_ready(this);
    System.out.println(this + " esta pronto.");
    boa.impl_is_ready();
  }
  public factory.ServerObject createServerObject(String sName) {
     return new ServerObject(sName);
  }
  public static void main(String[] args) {
     FactoryServer  server = new FactoryServerImpl("FactoryServer", args);
  }
}
```

Classe ServerObject

```
package factory;
import java.io.Serializable;
public final class ServerObject implements Serializable{
```

```
  private final String _sName;
  public ServerObject(String sName) {
     _sName = sName;
  }
  public String getName() {
     return _sName;
  }
}
```

- **Conseqüência:** fornece uma forma de criar objeto remoto mais flexível do que criar um objeto diretamente.

- **Usos conhecidos:** um padrão factory é utilizado de várias maneiras: é utilizado em aplicações não distribuídas, para centralizar a instanciação de objeto e em inúmeras aplicações distribuídas.

4.1.2. Padrão Observer

- **Nome**. Observer.

- **Intenção**: uma exigência comum de sistemas distribuídos é que eles possuam conhecimento relativo ao estado de um objeto remoto. Constantemente, conferindo mudanças de um objeto remoto, consome recursos de cliente, recursos de servidor e bandwidth (banda larga). O padrão observer permite a notificação de cliente em mudanças de servidor.

- **Motivação:** este padrão de projeto é utilizado em muitos ambientes onde o objeto-cliente necessita constantemente de conhecimentos de mudanças de valor do objeto-servidor. O padrão observer permite que o cliente sincronize com o servidor um valor mínimo até um certo ponto. Como o tráfico na rede existe somente quando um

valor é mudado, a banda-larga não será usada em vão. Ademais seus recursos serão usados mais eficientemente, porque clientes não serão constantemente solicitados por pedidos de mudanças ao servidor.

- **Aplicabilidade:** conforme a utilização do objeto-servidor, pelo padrão observer, gastaria-se algum tempo notificando os clientes da mudança de estado do servidor. Ao implementar o objeto-servidor, são necessárias decisões sobre a sincronização das mudanças e a maneira pela qual são alocados nos clientes. Em algumas situações o servidor irá notificar o cliente antes de fazer as mudanças e atualizações em si mesmo. Em outras situações, o servidor irá refletir sobre as mudanças internas e somente depois mandará a notificação ao cliente em um tread separado (baixa prioridade). Adicionalmente, desde que dois modos de comunicação entre cliente e servidor sejam requeridos, deve-se assegurar que existe algum dispositivo de segurança externo que permita isto. Se um firewall protege um cliente de receber métodos de invocação, este padrão não pode ser usado.

- **Estrutura**:

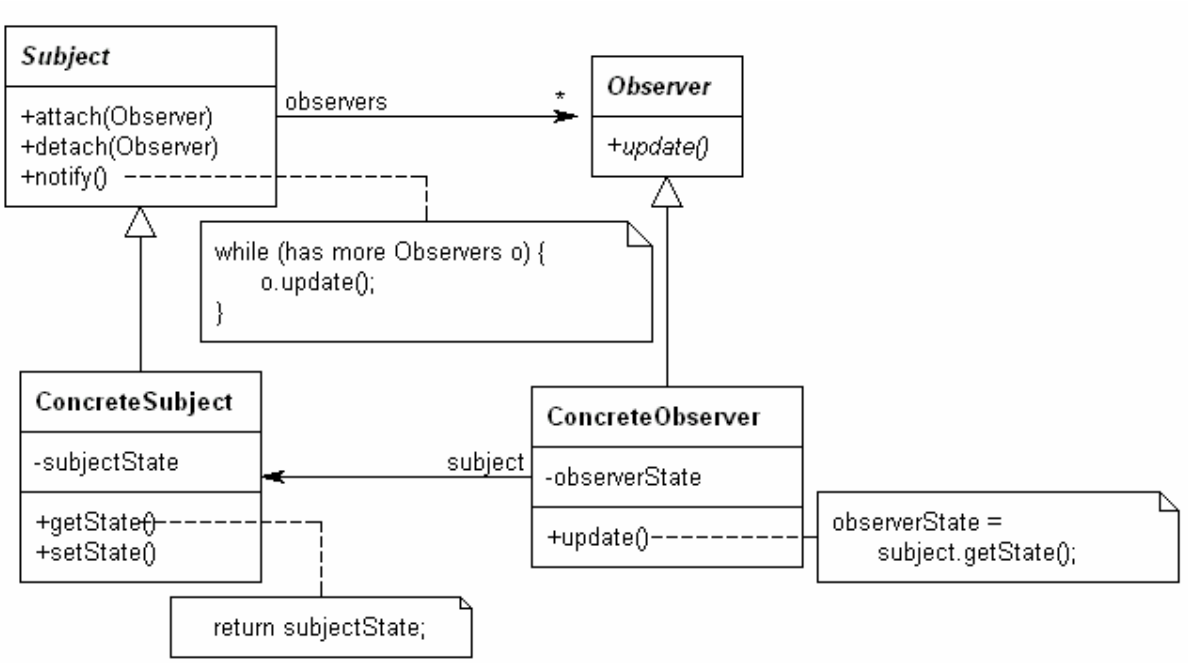

Figura 14 - Estrutura padrão observer.

- **Problema:** a função de um objeto-cliente é representar algum estado em correspondência com o objeto-servidor ou então entrar em ação em caso de mudança do objeto-servidor. Se o objeto-cliente tiver de ser notificado imediatamente de mudanças do estado do objeto-servidor, há duas soluções possíveis para notificar o problema: o objeto-cliente poderia conferir as mudanças para as várias unidades; ou, ainda, o servidor poderia notificar o cliente somente quando a mudança ocorresse. Havendo clientes constantemente sendo checados sobre mudanças do servidor, existe também um dreno em recursos (ambos do cliente e servidor), que vão requerer que a banda larga (bandwidth) seja dedicada a tal verificação. O padrão observer discute uma maneira lógica pela qual os clientes podem receber notificações das mudanças de estado do servidor.

- **Solução:** o padrão observer funciona até certo ponto semelhantemente à linguagem Java, utilizando delegação evento modelo. Sob o padrão observer, os clientes expõem um método (por uma interface) que é invocado para indicar uma mudança no objeto-servidor. O cliente registra o servidor como um ouvinte interessado. Quando mudanças ocorrem no objeto-servidor, este envia informações sobre mudanças de todos os clientes.

- **Exemplo:** a seguir exemplos demonstram uma simples aplicação em que um cliente registra interesse com um ponto quote server. O cliente notifica o quote servidor de todos os símbolos interessados e o servidor notifica o cliente sempre que um desse avalia mudanças. Há duas classes e uma interface contida nesta aplicação. O QuoteClientI interface identifica o método pelo qual o cliente obtém notificação de mudanças. A classe QuoteClient implementa o QuoteClientI interface e escuta mudanças para alguns símbolos. A classe QuoteServer rastreia todos, ouvindo e enviando notificações sempre que um registro símbolo tem um valor modificado.

Classe QuoteClientI Interface

```
// Interface a ser implementada por todos os objetos interessados em receber
   valores em mudanças de eventos
public interface QuoteClientI {
   public void quoteValueChanged(String sSymbol, double dNewValue); }
```

Classe QuoteClient

```
// A classe QuoteClient registra o interesse do servidor em localizar valores de diferentes pontos. Sempre que o servidor detectar uma mudança, notificará o quoteClient invocando o método quoteValueChanged()
import java.util.*;
public final class QuoteClient implements QuoteClientI {
    private Hashtable    hshPortfolio;
    public QuoteClient() {
                    _hshPortfolio = new Hashtable();
                    regWithServer();     }
// Registra no servidor  interesse em receber notificação quando houver mudanças de valores.
private final void regWithServer()  {
        org.omg.CORBA.ORB orb = orb.omg.CORBA.ORB.init(args,null);
       QuoteServer  server =  QuoteServerHelper.bind(orb, "QuoteServer");
        server.regListener("INKT",  this);
        server.regListener("MOBI", this);
        server.regListener("NGEN",  this);
        server.regListener("ERICY",  this);
        _hshPortfolio.put("INKT",  new Double(0));
        _hshPortfolio.put("MOBI",  new Double(0));
        _hshPortfolio.put("NGEN",  new Double(0));
        _hshPortfolio.put("ERICY",  new Double(0));  }
// Invocado sempre  que um valor estiver associado a um símbolo interessado em mudança
public void quoteValueChanged(String sSymbol,  double dNweValue)  {
        //  mostra as mudanças
        System.out.println ("\n");
        System.out.println (sSymbol+" changed value");
        System.out.println ("old value: " + _hshPortfolio.get(sSymbol));
        System.out.println ("new value: " + dNewValue);
        //  armazena o novo valor
        _hshPortfolio.put(sSymbol,  new Double(dNewValue));
        }
        public static void main(String[]  args)  {
            QuoteClient  client  =  new  QuoteClient();}
}
```

Classe QuoteServer

```
// A classe QuoteServer monitora e alimenta pontos notificando as partes interessadas quando ocorrerem mudanças em valores de símbolos registrados.

import java.util.*;

public final class QuoteServer {

        // Listas são armazenadas em tabelas ou vetores. A tabela usa uma chave para registrar os símbolos
            com um valor um vetor armazena sua lista.

        Private Hashtable   _hshListeners;

        Public  QuoteServer()  {

         org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);

         org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();

         boa.obj_is_ready(this);

         System.out.println(this + " esta pronto.");

         boa.impl_is_ready();

        _hshListeners   =  new  Hashtable();

        }

// Envia mudanças de valores para todas as listas.  A maneira que o  objeto QuoteServer monitora
os pontos de mudança está além  da extensão deste padrão. Simplesmente assume-se que o método é invocado quando necessário.

        private void sendChangeForSymbol(String sSymbol,  double dNewValue)  {

              //  checa se há um ouvinte para este símbolo

                Object  o  =  _hshListeners.get(sSymbol);

                if (o  != null)  {

                   Enumeration listeners  =  ((Vector)o).elements();

                   While(listeners.hasMoreElements())  {

                   ((QuoteClientI)listeners.nexElement()).quoteValueChanged(sSymbol,  dNewValue);

                         }

                  }

      }

      // Invocado por clientes para registrar interesse em um servidor para um específico símbolo.

        public void regListener(String sSymbol,  QuoteClientI  client)  {

        Object  o  =  _hshListeners.get(sSymbol);

        If (o  !=  null)  {

           ((Vector)o).addElement(client);

        }

        else  {  //  cria o vetor

                Vector vecListeners  =  new  Vector();
```

```
                vecListeners.addElement(client);
                HshListeners.put(sSymbol,  vecListeners);
            }
        }
}
```

- **Conseqüência:** a) com a notificação em mudanças no servidor aos clientes interessados, diminui consumo de recursos do cliente e servidor; b) reduz tráfico na rede; c) clientes não podem ter firewall instalado.

- **Usos conhecidos:** o padrão observer tem um trabalho paralelo quando da busca de informações. Geralmente envolve pegar um conteúdo de alguma tabela, originando informações para um ouvinte. Por exemplo, ao invés de checar todo dia o site na Web The New York Times, o próprio site entrega suas informações diretamente no seu desktop sempre que ocorrer alguma mudança.

4.1.3. Padrão Callback

- **Nome:** Callback.

- **Intenção:** o papel de um objeto-servidor é freqüentemente de executar alguns negócios lógicos que não podem ser realizados pelo objeto-cliente. Assumindo que este processo leva um tempo significante para ser executado, um cliente pode não ficar esperando que o método solicitado ao servidor seja completado para continuar com o processamento. Como alternativa, o objeto-servidor pode retornar imediatamente um valor nulo do método solicitado, liberando o cliente para outras ações, enquanto o servidor fica executando o negócio para depois que terminar o processo, enviar o resultado para o cliente utilizando a mesma thread.

- **Motivação:** nestes dois caminhos de comunicação entre cliente e servidor, deve ser assegurado que dispositivo de segurança externo o permitira.

- **Aplicabilidade:** este padrão é aplicável em muitos ambientes nos quais a resposta de processamento do servidor para o pedido de um cliente irá demorar muito.

- **Estrutura**:

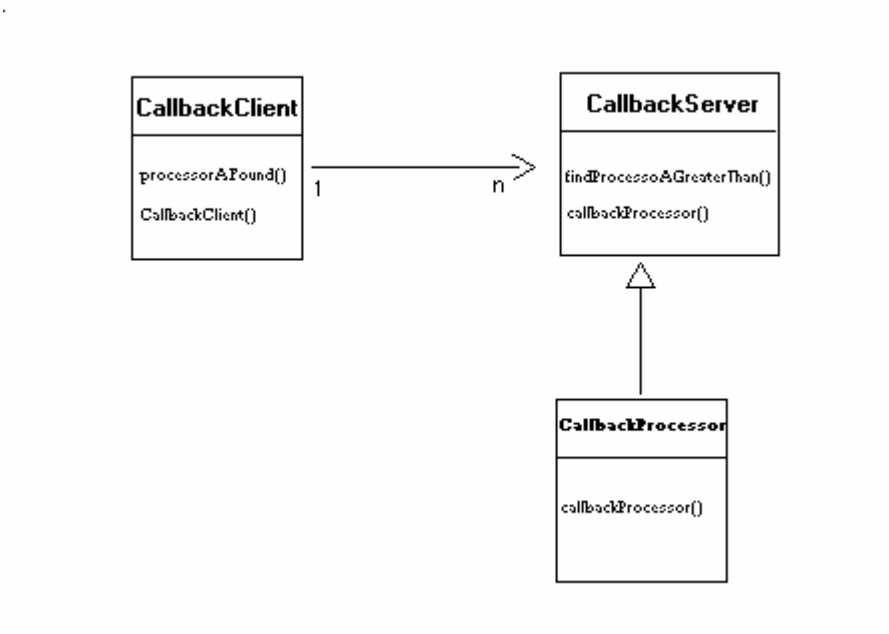

Figura 15 - Estrutura padrão callback.

- **Problema:** é comum para um objeto-cliente pedir algum dado para o objeto-servidor. Assumindo que o processo leva somente um ou dois segundos, o cliente não precisa se preocupar com o tempo de processamento envolvido. Contudo, se o servidor levar 10, 15 120, ou mais segundos, o cliente poderia ter que ficar esperando um tempo muito longo pelo retorno do valor executado pelo método. Se a espera do cliente por um retorno de valor de um determinado método for muito longa, podem ocorrer bloqueios ou uma excessiva lentidão na rede, o que não é, obviamente, uma situação desejável. Adicionalmente, dependendo da tecnologia usada para habilitar a

computação distribuída, um intervalo poderia acontecer se o objeto de servidor necessitar de muito tempo para devolver um valor.

- **Solução:** as funções do Padrão Callback podem permitir que o cliente emita um pedido ao servidor e então ter um retorno imediato, sem que tenha sido efetivado o processamento do pedido. O objeto-servidor então processa o pedido e passa o resultado para o cliente. Em muitas situações, o servidor executa todo o processamento em um thread separado para permitir uma conexão adicional. Se um firewall existir no cliente, para não permitir invocações de métodos, este padrão não poderá ser utilizado.

- **Exemplo:** neste exemplo será implementada uma classe servidora que tenha a capacidade de executar um processo e retornar o resultado em uma mesma thread quando o mesmo terminar. São necessárias três classes para compor este exemplo: CallbackClient liga para o servidor solicitando um serviço; CallbackServer fica esperando o pedido do cliente e, quando o recebe, usa uma outra thread para solicitar o processamento para a classe CallbackProcessor; a classe CallbackProcessor executa o processo e retorna um valor correto ou nulo.

Classe CallbackClient

```
// A classe CallbackClient irá fazer a ligação com o objeto-servidor pedindo a execução de um processo qualquer e então responderá o resultado quando o servidor prover.
package callback;
import java.io.Serializable;
public final class CallbackClient implements Serializable{
 public CallbackClient(String[] args) {
    // Ativa o ORB na rede para fazer conexão com servidor e esperar retorno
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    CallbackServer server = CallbackHelper.bind(orb, "CallbackServer");
    server.findProcessoAGreaterThan(Y, this);
 }
 public void processoAFound(Long lValue) {
```

```
    if (lValue == null) System.out.println("Processo not found");
    else System.out.println("ValorProcesso Found: "+lValue);
  }
  public static void main(String[] args) {
    CallbackClient cliente = new CallbackClient(args);
  }
}
```

Classe CallbackServer

```
package callback;
public final class CallbackServer extends _CallbackImplBase{
  public CallbackServer(String name) {
    super(name);
  }
  public void findProcessoAGreaterThan(int lBase, CallbackClient client) {
     CallbackProcessor processor = new CallbackProcessor(lBase, client);
     processor.start();
  }
  class CallbackProcessor extends Thread {
    private long          _lBase;
    private CallbackClient _client;

    public CallbackProcessor(long lBase, CallbackClient client) {
      _lBase = lBase;
      _client = client;
    }

    public void run() {
      long lFoundValue = Y1 ; // processo a ser executado (Calculo)
      _client.processoAFound(new Long(lFoundValue));
    }
  }
  public static void main(String[] args) {
    //Conexão com Corba para ligar o servidor
    CallbackServer servidor = new CallbackServer("CallbackServer");
```

```
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
    boa.obj_is_ready(servidor);
    System.out.println(servidor + " esta pronto.");
    boa.impl_is_ready();
  }
}
```

- **Conseqüência:** a) libera cliente para outros processos; b) firewall não pode estar habilitado.

- **Usos Conhecidos:** uma situação comum que requer o uso de um padrão callback ocorre quando é requerida a execução de um processo em um sistema que leva um longo tempo para gerar uma resposta.

4.1.4. Padrão State

- **Nome:** State.

- **Intenção:** permite a um objeto alterar o seu comportamento quando seu estado interno muda. O objeto parecerá ter alterado sua classe.

- **Motivação:** a idéia chave deste padrão é introduzir uma classe abstrata para representar os estados. A classe declara uma interface comum para todas as outras que representam diferentes estados operacionais. As subclasses implementam comportamentos específicos ao estado.

- **Aplicabilidade:** use o padrão State em algum dos dois casos seguintes:

    - o comportamento de um objeto depende do seu estado e ele pode alterar o seu comportamento em tempo de execução, dependendo desse estado;

- as operações têm comandos condicionais grandes, de várias alternativas e que dependem do estado do objeto. Este estado é normalmente representado por uma ou mais constantes enumeradas. Freqüentemente, várias operações conterão esta mesma estrutura condicional. O padrão State coloca cada ramo do comando adicional em uma classe separada. Isto lhe permite tratar o estado do objeto como um objeto propriamente dito, o qual pode variar independentemente dos outros.

- **Estrutura**:

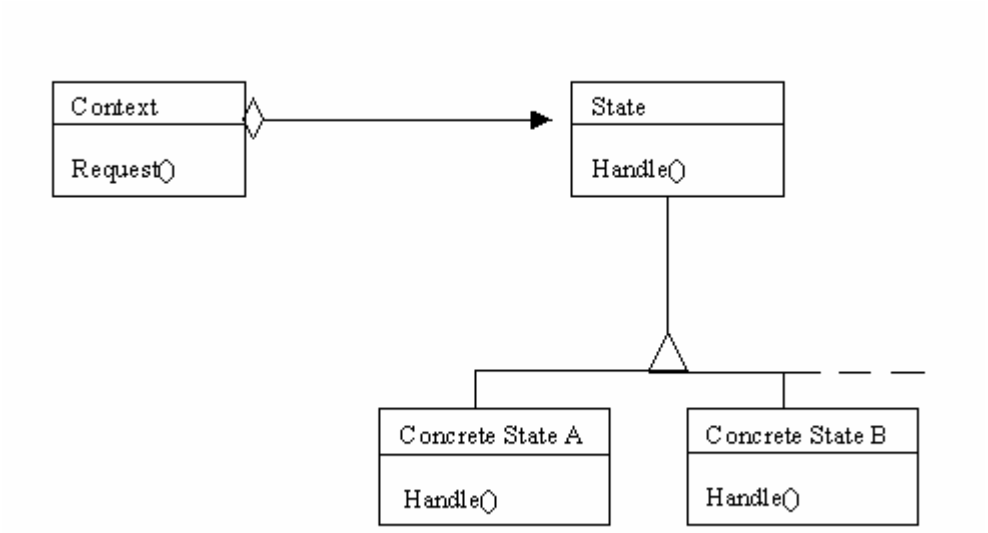

Figura 16 - Estrutura padrão state.

- **Problema**: o comportamento de um objeto depende do seu estado. Testes de condições são indesejáveis por causa da complexidade, da escalabilidade ou da duplicidade.

- **Solução:** criar uma classe para cada estado que influencie o comportamento do objeto dependente do estado. Baseados no polimorfismo, criam-se métodos para cada classe que representa um estado para tratar o comportamento do objeto contexto. Quando é recebida uma mensagem dependente de estado pelo objeto contexto, repasse-a para o objeto estado.

- **Exemplo**:

Classe EstadoFind

```
Public abstract class EstadoFind {
Public abstract void processo();
}
```

Classe EstadoA

```
public class EstadoA extends EstadoFind {
public EstadoA() {
}
public processo() {
// especifica o processo a ser executado
}
}
```

Classe EstadoB

```
public class EstadoB extends EstadoFind {
public EstadoA() {
}
public processo() {
// especifica o processo a ser executado
}
}
```

- Conseqüência: a) minimiza testes de condições; b) minimiza complexidade; c) aplicação das características do polimorfismo.

- **Usos conhecidos:** a maioria dos programas populares que fazem desenhos interativos fornecem ferramentas para a execução de operações através de manipulação direta. Por exemplo, uma ferramenta de seleção que permite ao usuário selecionar formas: o usuário vê esta atividade como se fosse simplesmente escolher uma ferramenta e utilizá-la, mas, na realidade, o comportamento no desenvolvimento muda de forma a cada nova seleção utilizada. Isto significa dizer que este padrão poder estar presente em qualquer software que possibilite a seleção de opções através de botões, menus etc.

4.1.5. Padrão Singleton

- **Nome**: Singleton.

- **Intenção:** garantir que uma classe tenha somente uma instância e fornecer um ponto global de acesso para a mesma.

- **Motivação:** é importante para algumas classes ter uma, e apenas uma, instância. Por exemplo, embora possam existir muitas impressoras em um sistema, deveria haver somente um spooler de impressoras ou então um sistema de contabilidade dedicado a servir somente a uma companhia.

- **Aplicabilidade:** use o padrão Singleton quando:

  - deveria haver apenas uma instância de uma classe, e essa instância deve garantir acesso aos clientes através de um ponto bem conhecido;

  - a única instância tiver de ser extensível através de subclasses, possibilitando aos clientes usarem uma instância estendida sem alterar o seu código.

- **Estrutura**:

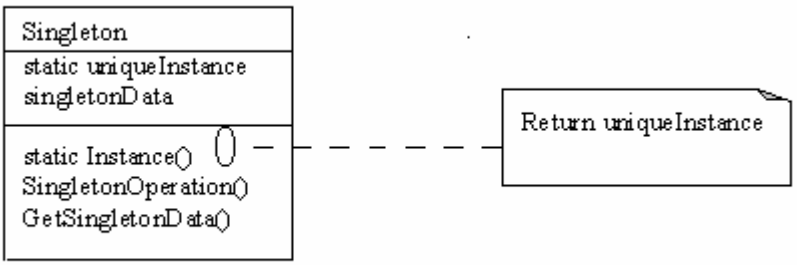

Figura 17 - Estrutura padrão singleton.

- **Problema:** é permitida exatamente uma única instância de uma classe. Os objetos necessitam de um único ponto de acesso.

- **Solução:** definir um método de classe ou uma função estática.

- **Exemplo:**

Classe Loja

```
Public class Loja implements Serializable {
Private static Cliente instancia;
Public Loja() {
}
public static Cliente instancia() {
  if (instancia == null)
     instancia = new Cliente();
  return instancia;
}
}
```

- **Conseqüência:** a) garante instanciação única de uma classe; b) permite controle total sobre como e quando os clientes acessam.

- **Usos conhecidos:** o toolkit para construção de interfaces de usuário InterViews [LINT92] usa o padrão Singleton para acessar as únicas instâncias de suas classes Session e WidgetKit, entre outras. Um exemplo mais sutil é o relacionamento entre classes e suas metaclasses. Uma metaclasse é a classe de uma classe e cada metaclasse tem uma instância. As metaclasses não têm nomes, mas registram e acompanham a sua única instância e, normalmente, não criarão outra.

4.2. Nível Arquitetura Global

O nível global compreende múltiplos empreendimentos. O principal assunto, enfoca o limite dos empreendimentos e o seu impacto nos softwares desenvolvidos. O nível global envolve linguagens padrões e uma política que afeta milhares de empreendimentos. Definir limites em um sistema global, se não impossível, é muito difícil. O nível global deve prover um protocolo que beneficie organizações afim de permitir meios de interoperabilidade e comunicação para diferentes empreendimentos.

O melhor exemplo para um sistema global é a Internet, que pode ser definida como uma coleção de padrões relacionados e como uma política que existe no mundo todo para permitir o compartilhamento de informações. Um importante aspecto da Internet é que uma coleção padrão pode ser disponibilizada para qualquer pessoa que desejar compartilhar e acessar informações de outras organizações.

4.2.1. Padrão CORBA-JSP-Gateway

- **Nome**: CORBA-JSP-Gateway.

- **Intenção**: usar CORBA como uma fonte de informação para a geração de páginas HTML.

- **Motivação:** administração de mudanças.

- **Aplicabilidade:** se for necessário integrar aplicações existentes com a intranet ou na WEB ou se desejar acessar informações de sistemas legados através de browsers na Web ou, ainda, usar o CORBA como um mecanismo de infra-estrutura para integração de aplicação.

Estrutura:

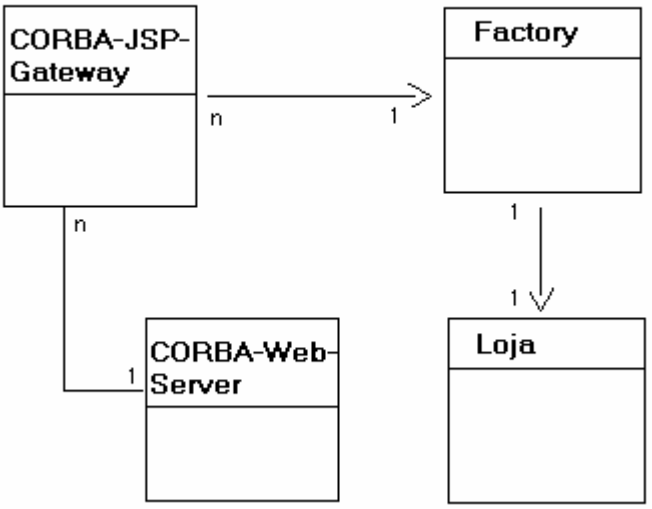

Figura 18 - Estrutura padrão CORBA-JSP-Gateway.

- **Problema**: para acessar informações baseadas na Web, um usuário entra ou seleciona uma ligação em um browser ou entra com informações em forma de HTML e ativa um URL de HTTP. O browser junta a informação em uma série de pares de valores-nomes e envia isto ao servidor de HTTP indicado na string URL. A URL pode identificar um documento HTML no servidor, um arquivo, ou um script à ser executado. O HTTP empacota as informações em um arquivo HTML, ou de saída de um programa no caso de um script e transporta as informações de volta ao browser. O mecanismo é simples e eficiente. Porém, para a maior parte, a interface é usada para exibir informação estática armazenada em um arquivo previamente autorizado em HTML. Alguns scripts para construção de páginas dinâmicas estão disponíveis, mas eles geralmente são de difícil codificação para que se possa localizá-los na máquina de HTTP ou provêem referências a fontes de informações específicas, limitando a flexibilidade do sistema.

- **Solução**: uma URL é um pedido de informação pré-programado. No modelo da Web, selecionando uma URL, envia-se um pedido de informação para um servidor. JSP (JavaServer Pages) é parte do protocolo HTTP e permite a execução de programas externos e scripts em um servidor. Primeiro, cria-se uma página estática em HTML para um JSP script. Pode ser executado por um servidor HTTP. Este JSP script poderia ser um cliente CORBA, com a existência de um ORB na mesma máquina como o servidor HTTP. Especificando um cliente CORBA para ser invocado por intermédio de interfaces JSP, o cliente JSP script/CORBA pode processar o pedido acessando os serviços CORBA. Este cliente poderá receber o benefício da localização CORBA transparente, ligando dinamicamente a implementação de serviços, ativando automaticamente o servidor, e assim por diante. Isto é importante: enquanto a aplicação cliente deve ser localizada na mesma máquina como o HTTP, o CORBA service usado pelo cliente pode estar distribuído. Depois que os pedidos de CORBA são processados, o cliente CORBA junta as informações no servidor HTTP. Como a saída de um JSP script deve ser em uma página HTML, o cliente CORBA é responsável pela geração dinâmica de uma página ou por prover a localização de

uma página no servidor HTTP. O servidor HTTP envia um documento HTML para o cliente como resposta ao pedido inicial.

- **Exemplo**:

Classe Gateway

```
package aplicacao.corba;
import aplicacao.balcao.*;
import org.omg.CORBA.ORB;
public final class Gateway {
  private Loja loja;
  private Balcao balcao;
  private IFactoryServer fserver;
  private IWebServer wserver;
  private ORB orb;
  public Gateway(String id) {
    this(id, new String[]{});
  }
  public Gateway(String id, String[] args) {
    orb = ORB.init(args,null);
    fserver = IFactoryServerHelper.bind(orb, "FactoryServer");
     wserver = IWebServerHelper.bind(orb, "WebServer");
    loja = fserver.criaLoja();
    balcao = loja.getBalcao();
   wserver.putBalcao(id, balcao);
    System.out.println(fserver._object_name() + " foi criado com sucesso.");
    System.out.println(wserver._object_name() + " foi criado com sucesso.");
    System.out.println(wserver.getBalcao(id) + " foi criado com sucesso.");
  }
   public static void main(String[] args) {
     Gateway client = new Gateway("1", args);
  }
}
```

FactoryServer

```
package aplicacao.corba;
import aplicacao.balcao.*;
public final class FactoryServer extends _IFactoryServerImplBase {
 org.omg.CORBA.ORB orb;
 org.omg.CORBA.BOA boa;
 public FactoryServer(String name) {
     super(name);
 }
 public void esperaPorConexao(String[] args) {
    orb = org.omg.CORBA.ORB.init(args,null);
    boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
    boa.obj_is_ready(this);
    System.out.println(this + " esta pronto.");
    boa.impl_is_ready();
 }
 public Loja criaLoja() {
    return Loja.instancia();
 }
 }
```

WebServer

```
package aplicacao.corba;
import aplicacao.balcao.*;
import java.util.Hashtable;
public class WebServer extends _IWebServerImplBase {
 private String _nome;
 private Hashtable htBalcao = new Hashtable();
 public WebServer(String nome) {
     super(nome);
     _nome = nome;
 }
 public void esperaPorConexao(String[] args) {
```

```
    org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
    org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
    boa.obj_is_ready(this);
    System.out.println(this + " esta pronto.");
    boa.impl_is_ready();
  }
  public Balcao getBalcao(String id) {
     return (Balcao) htBalcao.get(id);
  }
  public void putBalcao(String id, Balcao balcao) {
     htBalcao.put(id, balcao);
  }
  public void removeBalcao(String id) {
     htBalcao.remove(id);
  }
}
```

ServidorFactory

```
package aplicacao.corba;
public class ServidorFactory {
 public static void main(String[] args) {
     FactoryServer  server = new FactoryServer("FactoryServer");
     server.esperaPorConexao(args);
 }}
```

ServidorWeb

```
package aplicacao.corba;
public class ServidorWeb {
  public static void main(String[] args) {
     WebServer  server = new WebServer("WebServer");
     server.esperaPorConexao(args);
  }
}
```

- **Conseqüência:** a) não há acesso direto a objetos CORBA; b) o cliente CORBA tem que executar na mesma máquina que o HTTP server.

- **Usos Conhecidos:** este padrão é uma variante interessante para solução Gateway (Portal). Especificamente, poderia ser utilizado para que qualquer um provesse Gateway com a internet e outras tecnologias ou então como um modelo para empacotar as funcionalidades do cliente CORBA com uma tecnologia já existente. Gateway é uma solução mais geral para interoperabilidade entre modelos de objeto diferentes. O exemplo mostrado no padrão gateway ilustra como a solução CORBA-JSP-Gateway pode ser invertida para permitir que a Internet se torne uma fonte de informações para aplicações CORBA.

4.2.2. Padrão CORBA-Web-Server

- **Nome:** CORBA-Web-Server.

- **Intenção:** estender as capacidades de aplicação intranet ou aplicações WWW, em particular, as páginas da Web, usando CORBA para manter o estado dos objetos na Web.

- **Motivação:** administração de funcionalidades.

- **Aplicabilidade:** quando for necessária a retenção de informação do estado de uma aplicação de servidor ou a melhora do desempenho, limitando o tamanho de endereços(handles) URL, ou a implementação de páginas mais complexas na Web.

- **Estrutura:**

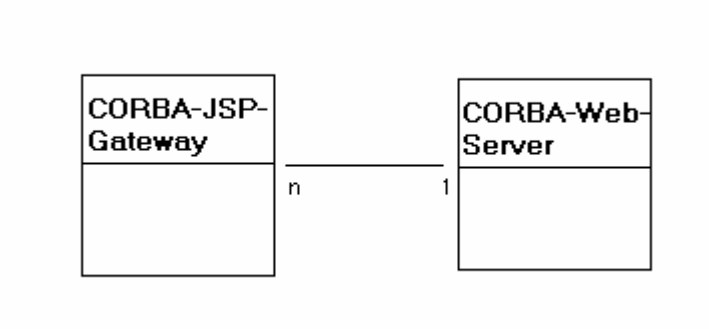

Figura 19 - Estrutura padrão CORBA-WEB-Server.

- **Problema:** assuntos sobre ciclo de vida são muito difíceis. Por exemplo: quando é direito excluir um objeto? O cliente precisa saber criar uma página HTML. O nome do objeto é passado de página para página.

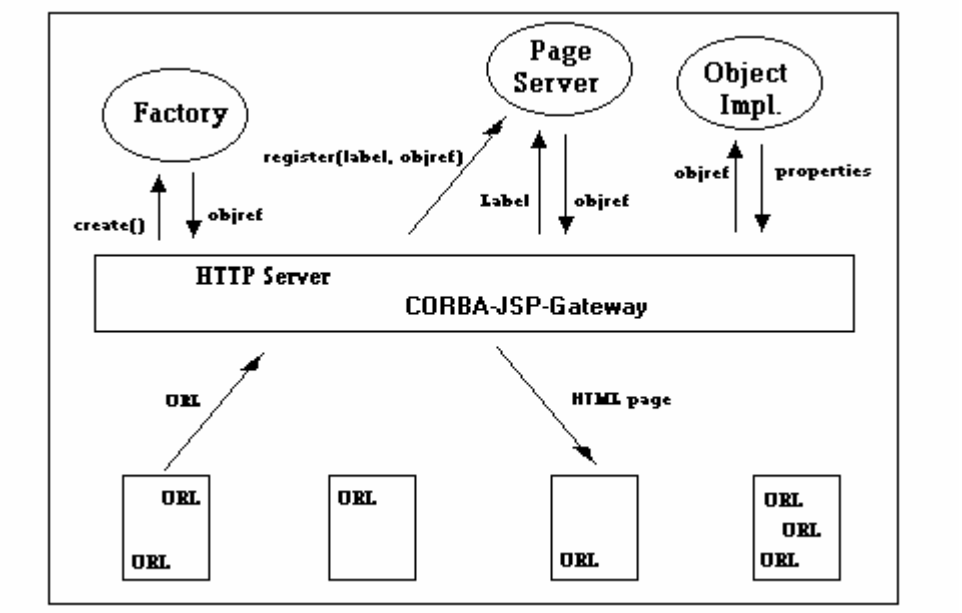

Figura 20 - O estado é mantido em objetos CORBA tendo um mapa de serviço da página como label para uma referência de objeto CORBA.

- **Solução:** primeiro, na chamada inicial para um cliente CORBA por intermédio da interface JSP, um símbolo deve ser criado para identificar o objeto atual em uso no ambiente CORBA. O símbolo ou pode ser um nome qualquer, um label, ou um campo string como referência. A grande vantagem de se usar este padrão é a limitação da quantia de informação passada para o browser, pois o campo string de referência do objeto pode ter um tamanho de até 1K bytes, não sendo recomendado como forma de símbolo. Este símbolo será passado como uma variável fixa na URL para cada página, usando uma interface de JSP. O símbolo é passado como uma variável escondida no fim da URL. Dependendo da aplicação, o nome pode ser conectado pela interface Forms ou gerado automaticamente em tempo de acesso de página. Quando um cliente CORBA é invocado, pode modificar o estado do objeto como um cliente padrão CORBA. Porém, não há necessidade de enviar uma coleção inteira de variáveis para representar o estado atual do objeto atrás do browser na Internet, ao invés de armazenar uma lista de variáveis escondidas no fim da URL para representar o estado do objeto atual, simplesmente envia-se o símbolo para o objeto. Quando um cliente CORBA é invocado novamente, o símbolo é usado para identificar o objeto CORBA atual, e a sua referência de objeto é retornada da OMG Naming Service, ou então a implementação é responsável para armazenar no mapa o símbolo da referência do objeto. Uma vez obtida a referência de objeto, o cliente CORBA pode invocar o objeto atual, embora sua localização e retorno de informação atual, incluam algum atributo ou propriedade que foram modificadas pelo clientes CORBA que tiveram acesso ao mesmo objeto. Além disso, quaisquer mudanças feitas pelo cliente CORBA no início, podem ser preservadas, podendo ser utilizadas através de invocações subseqüentes a clientes CORBA. Além disso, qualquer cliente CORBA que é invocado pelas interfaces de JSP tem acesso a qualquer objeto CORBA conhecido em tempo de compilação. Por exemplo: CORBA services; CORBA Naming Service da OMG; OMG Trader Service, ou um Factory. Agora, o ambiente da Web só precisa armazenar o símbolo para o objeto. Se o símbolo for um nome significante ou um label, poderá ser exibido como parte da criação dinâmica de uma página de HTML ou como um valor em uma interface Forms. Em cada chamada da interface JSP, clientes CORBA podem usar o nome para retornar o objeto e utilizar a interface objeto para armazenar e retornar informações sobre o seu estado. Se a

OMG Naming Service é usada para associar o símbolo a um objeto de referência CORBA, então existe um pequeno overhead; contudo, a OMG Naming Service pode vir a ser povoada com objetos de referência sem meios para determinar o fim do seu ciclo de vida. Então, freqüentemente, é desejável criar-se um serviço separado para armazenar referências de objeto usado para manter o estado de objetos usados para serviços de browsers assim seus ciclos de vida podem ser administrados através de serviços separados (especificamente se não teve acesso por um prazo especificado). O Padrão de Atributos Dinâmicos é um método útil para armazenar atributos de objetos associados a uma referência de objeto, como é o caso do OMG Property Service.

- **Exemplo:**

Classe Gateway

```
package aplicacao.corba;
import aplicacao.balcao.*;
import org.omg.CORBA.ORB;
public final class Gateway {
 private Loja loja;
 private Balcao balcao;
 private IWebServer wserver;
 private ORB orb;
 public Gateway(String id) {
   this(id, new String[]{});
 }
 public Gateway(String id, String[] args) {
   orb = ORB.init(args,null);
    wserver = IWebServerHelper.bind(orb, "WebServer");
   wserver.putBalcao(id, balcao);
   System.out.println(fserver._object_name() + " foi criado com sucesso.");
   System.out.println(wserver._object_name() + " foi criado com sucesso.");
   System.out.println(wserver.getBalcao(id) + " foi criado com sucesso.");
 }
  public static void main(String[] args) {
    Gateway client = new Gateway("1", args);  }}
```

Classe WebServer

```
package aplicacao.corba;
import aplicacao.balcao.*;
import java.util.Hashtable;
public class WebServer extends _IWebServerImplBase {
 private String _nome;
 private Hashtable htBalcao = new Hashtable();
 public WebServer(String nome) {
    super(nome);
    _nome = nome;
 }
 public void esperaPorConexao(String[] args) {
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
   org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
   boa.obj_is_ready(this);
   System.out.println(this + " esta pronto.");
   boa.impl_is_ready();
 }
 public Balcao getBalcao(String id) {
    return (Balcao) htBalcao.get(id);
 }
 public void putBalcao(String id, Balcao balcao) {
    htBalcao.put(id, balcao);
 }
 public void removeBalcao(String id) {
    htBalcao.remove(id);
 }
 public static void main(String[] args) {
    WebServer  server = new WebServer("WebServer");
    server.esperaPorConexao(args);
 }}
```

- **Conseqüência:** 1) o cliente precisa saber criar uma página HTML; 2) o nome dos objetos é passado de página para página.

- **Usos Conhecidos:** CORBA-JSP-Gateway - este padrão é usado para invocar clientes CORBA que precisam ter acesso à informação do estado das páginas Web. Gateway: este padrão pode ser substituído como meio de acessar informações de um ambiente discrepante.

4.2.3. Padrão DII-Web-Server

- **Nome:** DII-Web-Server.
- **Intenção:** utilizar as facilidades do CORBA DII para o Web browser.
- **Motivação:** administração de funcionalidades.
- **Aplicabilidade:** quando for necessário invocar interfaces dinamicamente da Web.
- **Estrutura:**

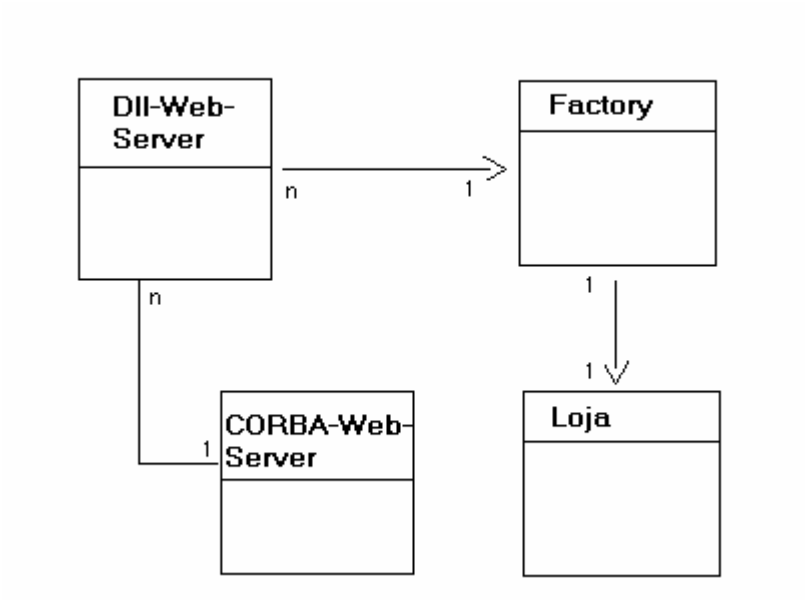

Figura 21 - Estrutura padrão DII-Web-Server.

- **Problema:** em uma página Web, há a necessidade de se colocar imagens a disposição do usuário para sua escolha. Portanto a utilização do acesso estático pode ser uma dificuldade. Através do padrão DII-Web-Server, pode-se utilizar o repositório do ambiente CORBA para colocar a disposição imagens, objetos e interfaces para escolha do usuário.

- **Solução:** uma interface Web deve ser apresentada de maneira que permita ao cliente, via browser, o acesso ao repositório de interface do ambiente do CORBA. Serão exibidos em tempo real o conteúdo do repositório de interface, onde podem estar imagens, classes de objetos ou páginas estáticas para serem utilizadas. A interface na página Web precisa permitir ao cliente selecionar qual dos arquivos deseja utilizar através de comunicação com aplicações CORBA. Logo um processo de seleção deve ser criado de maneira a suportar a interface desejada pelo cliente, possibilitando a entrada dos parâmetros necessários para a comunicação CORBA. Este padrão não é muito aceito, pois é necessário criar um processo de autorização para utilização do repositório, afim de que as informações disponibilizadas pelo processo não sejam empregadas de maneira errônea.

- **Exemplo:**

Classe DII-Gateway

```
package aplicacao.corba;
import aplicacao.balcao.*;
import org.omg.CORBA.ORB;
public final class DIIGateway {
  private Loja loja;
  private Balcao balcao;
  private IFactoryServer fserver;
  private IWebServer wserver;
  private ORB orb;
  public DIIGateway(String id, String fServerNome, String wServerNome, String ServerMetodo, String
wServerMetodo) {
    orb = ORB.init(new String[]{},null);
    fserver = IFactoryServerHelper.bind(orb, fServerNome);
    org.omg.CORBA.Request request = fserver._request(fServerMetodo);
    request.set_return_type(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_objref));
    request.invoke();
    loja = (Loja)request.return_value().extract_Object();
    balcao = loja.getBalcao();
     wserver = IWebServerHelper.bind(orb, wServerNome);
```

```
    request = wserver._request(wServerMetodo);
    request.add_in_arg().insert_string(id);

request.add_in_arg().insert_TypeCode(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_objref));
    request.set_return_type(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_void));
    request.invoke();
    System.out.println(fserver._object_name() + " foi criado com sucesso.");
    System.out.println(wserver._object_name() + " foi criado com sucesso.");
    System.out.println(wserver.getBalcao(id) + " foi criado com sucesso.");
  }
  public Loja getLoja() {
     return loja;
  }
  public void setLoja(Loja lj) {
      loja = lj;
  }
  public Balcao getBalcao() {
     return balcao;
  }}
```

FactoryServer

```
package aplicacao.corba;

import aplicacao.balcao.*;

public final class FactoryServer extends _IFactoryServerImplBase {

  org.omg.CORBA.ORB orb;

  org.omg.CORBA.BOA boa;

  public FactoryServer(String name) {

     super(name);

  }

  public void esperaPorConexao(String[] args) {

     orb = org.omg.CORBA.ORB.init(args,null);

     boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();

     boa.obj_is_ready(this);

    System.out.println(this + " esta pronto.");

     boa.impl_is_ready();

  }

  public Loja criaLoja() {

     return Loja.instancia();

  } }
```

WebServer

```
package aplicacao.corba;
import aplicacao.balcao.*;
import java.util.Hashtable;
public class WebServer extends _IWebServerImplBase {
 private String _nome;
 private Hashtable htBalcao = new Hashtable();
 public WebServer(String nome) {
    super(nome);
    _nome = nome;
 }
 public void esperaPorConexao(String[] args) {
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
   org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
   boa.obj_is_ready(this);
   System.out.println(this + " esta pronto.");
   boa.impl_is_ready();
 }
 public Balcao getBalcao(String id) {
    return (Balcao) htBalcao.get(id);
 }
 public void putBalcao(String id, Balcao balcao) {
    htBalcao.put(id, balcao);
 }
 public void removeBalcao(String id) {
    htBalcao.remove(id);
 }}
```

ServidorFactory

```
package aplicacao.corba;
public class ServidorFactory {
 public static void main(String[] args) {
     FactoryServer  server = new FactoryServer("FactoryServer");
     server.esperaPorConexao(args);
 }}
```

ServidorWeb

```
package aplicacao.corba;
public class ServidorWeb {
  public static void main(String[] args) {
     WebServer  server = new WebServer("WebServer");
     server.esperaPorConexao(args);
  }
}
```

- **Conseqüência:** a) requer do cliente ter conhecimento específico sobre os processos dinâmicos; b) requer complexos processos no cliente.

- **Usos Conhecidos:** este é um padrão novo sem conhecimento de uso na internet.

5. ESTUDO DE CASO: PROJETANDO UMA APLICAÇÃO BALCÃO UTILIZANDO CATÁLOGO DE PADRÕES DE PROJETO DISTRIBUÍDO APLICADOS COM CORBA E JAVA NA WEB.

Este estudo de caso é parte de um sistema de balcão que pode ser utilizado em uma loja qualquer. Esta aplicação tem como objetivo armazenar as vendas diárias e efetuar pagamentos. Esta é uma aplicação já de várias soluções no mercado, mas nosso objetivo aqui é mostrar como trabalhar em um ambiente orientado a objeto utilizando padrões de projeto já existentes, inserindo computação distribuída em suas classes.

Esta aplicação foi escolhida por se tratar de um sistema de informação muito comum, onde desenvolvedores podem encontrar detalhes suficientes para aplicar estes métodos a novos projetos, formalizando assim o processo de reutilização em objetos.

Nós não abordaremos o processo de análise e o projeto orientado a objetos aplicados a este sistema por não ser objetivo desta dissertação, mas deixamos claro que seriam processos de grande importância.

No entanto, para o desenvolvimento das classes utilizadas neste projeto, utilizamos a notação UML (Unified Modeling Language), que está sendo adotada pela OMG, como linguagem padrão para especificações orientadas a objetos.

5.1. Arquitetura Clássica de Três Camadas

Em um projeto orientado a objeto a arquitetura de três camadas (three-tier architeture) é a mais recomendada pela sua qualidade singular de separação da lógica da aplicação em uma camada intermediária separada da aplicação e do armazenamento. A camada de apresentação é relativamente livre de processamento ligado à aplicação, onde

as janelas repassaram as solicitações de tarefas para uma camada intermediária que obtém as regras de negócios. A vantagem desta apresentação é a aplicação da lógica em componentes separados, viabilizando em muito a reutilização do software. A seguir uma visão da arquitetura utilizada no projeto.

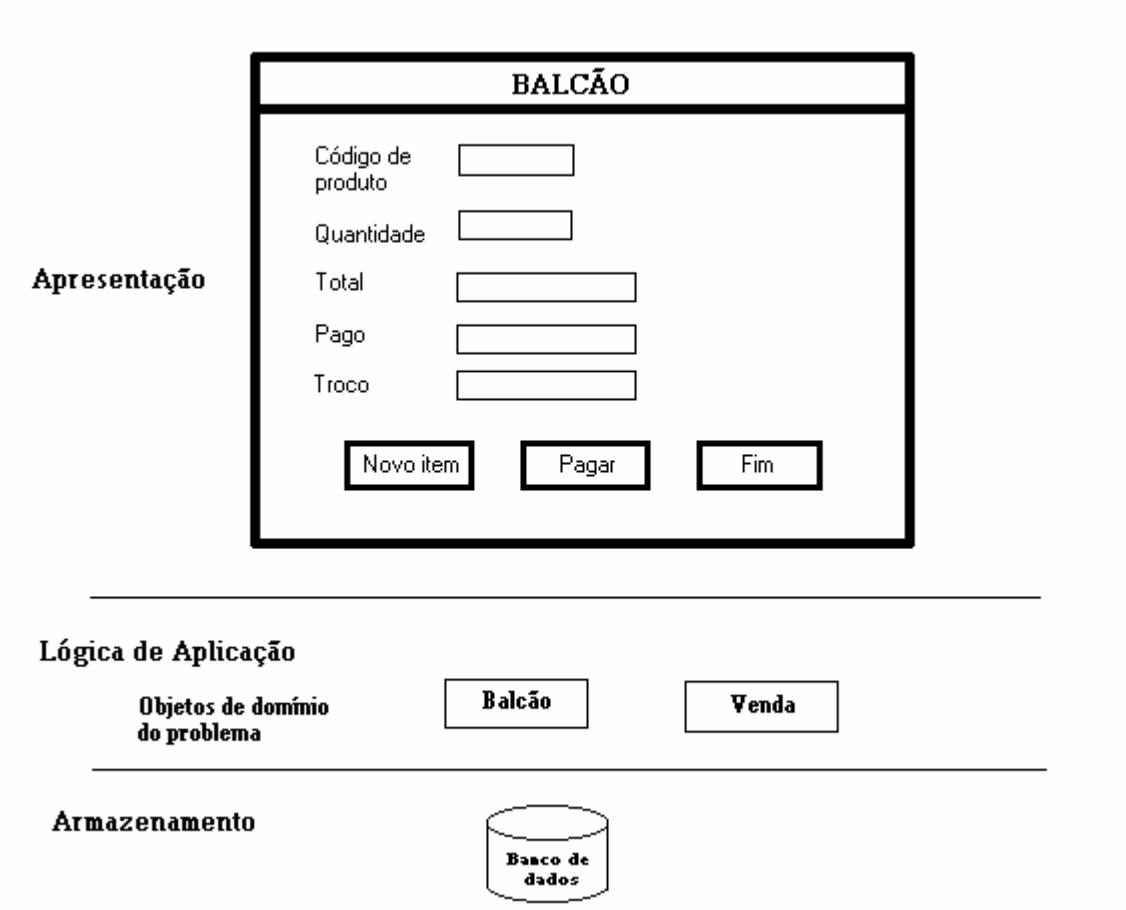

Figura 22 - Modelo do sistema em três camadas.

5.2. Modelo Conceitual do Domínio Balcão

O modelo da figura 23, tem como objetivo mostrar o domínio da regra de negócios da camada lógica da aplicação, a qual irá ser implementada em alguns padrões de projeto do catálogo proposto por esta dissertação.

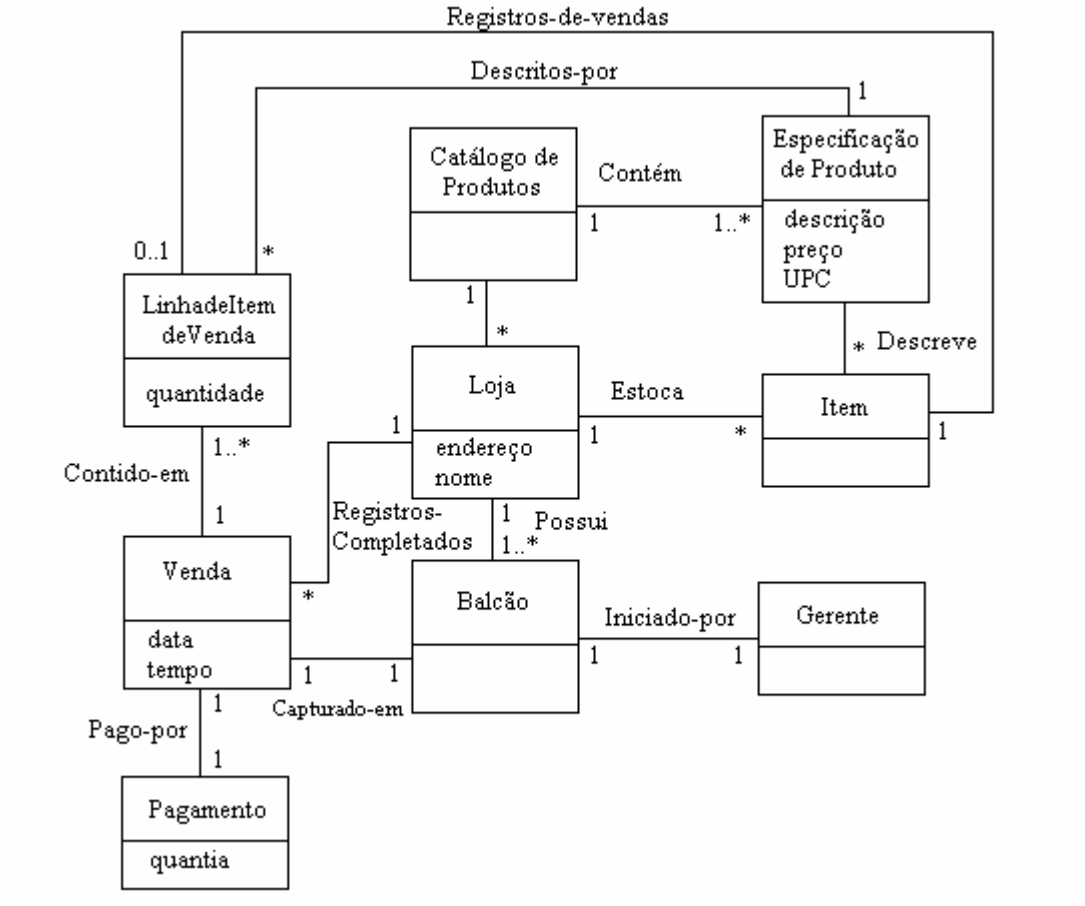

Figura 23 – Modelo conceitual para o domínio balcão.

5.3. Solução Programada em Java

Este capítulo não tem o objetivo de fazer deste projeto um programa Java robusto, por exemplo, com tratamento de exceções, acesso a base de dados, etc. Ele simplesmente tem como objetivo principal mostrar nas classes desenvolvidas onde foram incrementados alguns dos padrões de projetos mostrados no catálogo de padrões formalizados por esta dissertação.

As classes utilizadas são:

Classe Pagamento

```
Package balcao
Public class Pagamento
{
private float quantia;
public Pagamento (float dinheiroFornecido)
{
        this.quantia = dinheiroFornecido;
}
public float getQuantia()   { return quantia;}
```

Classe CadastrodeProdutos

```
Package balcao;
Import java.util.*;
Public class CadastrodeProdutos
{
private Hashtable especificacoesdeProdutos = new Hashtable();
public CadastrodeProdutos()
{
 EspecificaçãodeProduto ep =
        New EspecificacaodeProdutos.put (100, 1, “produto1”);
 EspecificacoesdeProdutos.put (new Integer (100), ep);
 Ep = new EspecificacaodeProduto (200, 1, “produto 2”);
 EspecificacoesdeProdutos.put (new Integer (200), ep);
```

```
}
public EspecificacaodeProduto getEspecificacao (int upc)
{
 return (EspecificacaodeProduto)
   especificacoesdeProdutos.get (new Integer (upc) );
}
}
```

Classe Balcao

```
Package post;
Import java.util.*;
Class BALCAO
{
private CadastrodeProdutos cadastrodeProdutos;
private Venda venda;
public BALCAO (cadastrodeProdutos cadastro)
{
 cadastrodeProdutos  =  cadastro;
{
public void terminarVenda ()
{
 venda.tornarseCompleta ();
}
public void entrarItem (int upc,  int  quantidade)
{
 if  (eNovaVenda () )
   {
     venda = new  Venda ();
  }
  EspecificacaodeProduto   espec =
         CadastrodeProdutos.especificacao (upc);
 Venda.construirLinhadeItem (espec, quantidade);
}
public void registrarPagamento (float  dinheiroFornecido)
{
 venda.efetuarPagamento (dinheiroFornecido);
```

```
}
private boolean eNovaVenda ()
{
  return (venda == null)  (venda.estaCompleta () );
}
}
```

Classe EspecificaçãodeProduto

```
Package post;
Public class EspecificacaodeProduto
{
private int  upc = 0;
private float preco = 0;
private String descricao = " " ;
public EspecificacaodeProduto  (int  upc,  float preco, String descricao)
{
this.upc = upc;
this.preco = preco;
this.descricao = descricao;
}
public int getUPC ()  { return upc ; }
public float getPreço ()  { return preco; }
public String getDescricao ()  { return descricao; }
}
```

Classe Venda

```
Package balcao;
Import java.util.*;
Class Venda
{
private Vector linhadeItens  =  new  Vector ();
private Date  data  =  new  Date ();
private boolean estaCompleta  = false;
private Pagamento pagamento;
public float getTroco ()
```

```
  {
    return pagamento.getQuantia () - total ();
  }
public void tornarseCompleta () { estaCompleta = true; }
public boolean estaCompleta () { return estaCompleta; }
public void construirLinhadeItem
   (EspecificacaodeProduto espec, int quantidade)
  {
   linhadeItens.addElement (
     new LinhadeItemdeVenda ( espec, quantidade) );
  }
public float total ()
  {
   float total = 0;
   Enumeration e = linhadeItens.elements ();
   While ( e.hasMoreElements () )
    {
      total += ( (LinhadeItemdeVenda) e.nextElement ( ) ) .subtotal ( );
    }
  return total;
  }
public void efetuarPagamento (float dinheiroFornecido)
  {
   pagamento = new Pagamento (dineiroFornecido);
  }
}
```

Classe LinhadeItemdeVenda

```
Package balcao;
Class LinhadeItemdeVenda
{
private int quantidade;
private EspecificacaodeProduto especdeProduto;
public LinhadeItemdeVenda
      (EspecificacaodeProduto espc, int quantidade)
{
 this.especdeProduto = espec;
```

```
  this.quantidade  =  quantidade;
}
public float subtotal ()
{
  return quantidade  *  especdeProduto.getPreço ();
}
}
```

Classe Loja

```
Package balcao;
Class Loja
{
private CadastrodeProdutos  cadastrodeProdutos
     =  new CadastrodeProdutos ();
private BALCAO balcao  =  new BALCAO (cadastrodeProdutos);
public BALCAO getBALCAO ()  (return balcao; )
}
```

5.4. Diagrama de Funcionalidades.

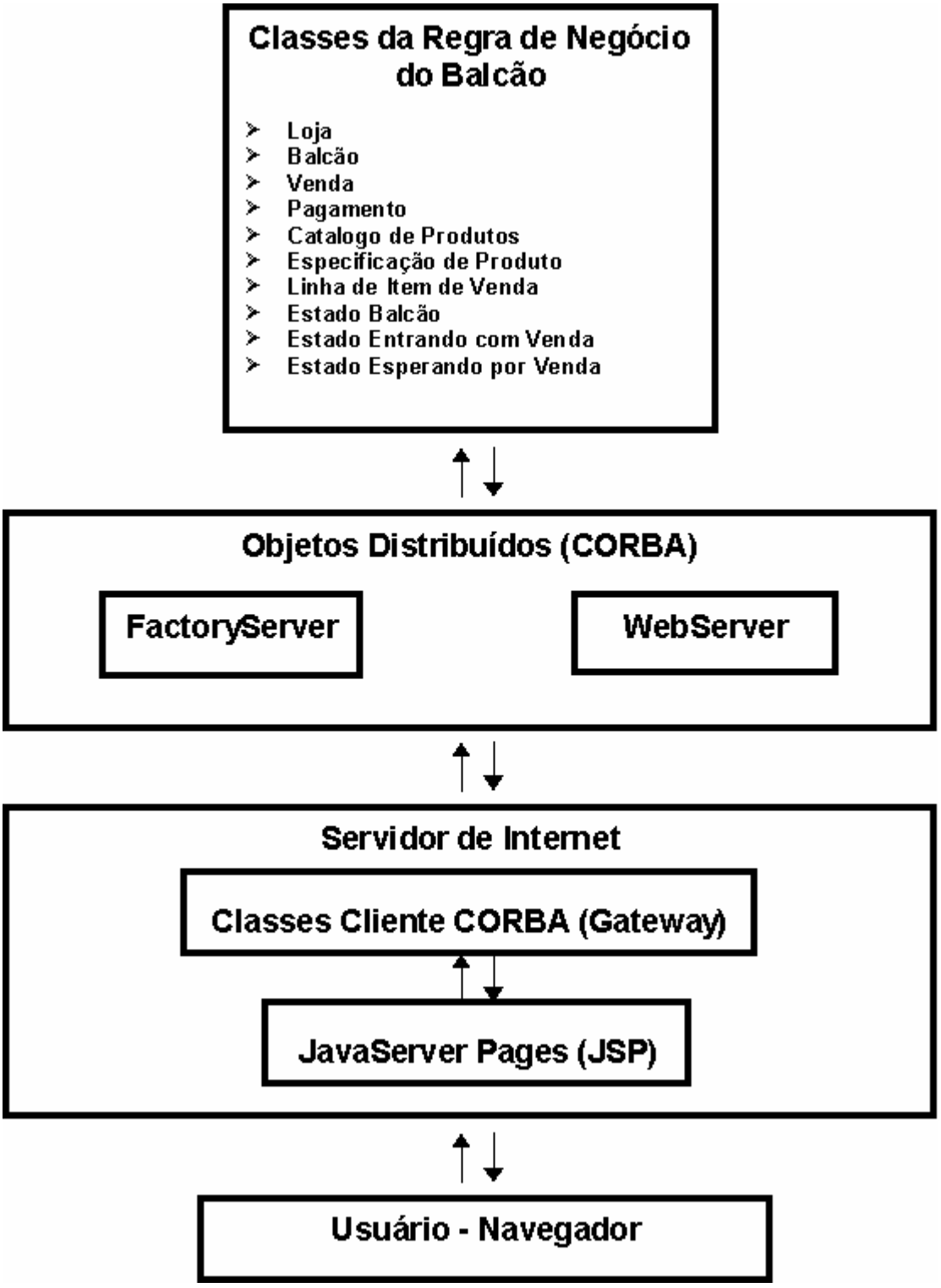

Figura 24 – Diagrama de funcionalidades.

5.5. Aplicando os Padrões nas Classes da Aplicação Balcão

Este capítulo enfatiza a utilização dos padrões de projetos catalogados no capítulo 4 para ajudar no projeto de sistemas orientados a objetos.

Com a explosão na utilização e desenvolvimento de aplicativos para internet, trabalhar com a arquitetura cliente/servidor tornou-se uma necessidade. O comércio eletrônico exige requisitos complexos para as transações executadas.

Existem algumas tecnologias, como a Active Server Page (ASP) e Common Gateway Interface (CGI) que ganharam popularidade dentro da comunidade de desenvolvimento da WEB. Atender os requisitos necessários para o desenvolvimento distribuído com interoperabilidade, transparência, estabilidade e eficiência no manuseio de solicitações de objetos múltiplos e simultâneos são questões e preocupações a serem consideradas.

O paradigma HTTP/CGI é muito lento, pelo fato de cada solicitação de acesso gerar um novo processo necessitando de memória residente. A ASP tem problema na estabilidade do servidor e em manusear objetos múltiplos.

Logo, a necessidade de se utilizar uma tecnologia que possa melhorar todos os requisitos acima expostos torna-se essencial.

O JavaServer Pages (JSP) é uma tecnologia baseada na linguagem Java, que simplifica o processo de desenvolvimento dinâmico de sites Web. O JSP funciona como um compartimento (container) que incorpora elementos dinâmicos.

O JSP é uma linguagem script que funciona no lado do servidor, ou seja, as páginas JavaServer são arquivos texto, normalmente com a extensão ".jsp" que substituem os arquivos estáticos HTML. As páginas JSP, além de utilizar objetos do servidor, podem incorporar e manipular objetos próprios, como Applets e Servlets. O JSP veio para integrar todas essas tecnologias escrita com Java puro.

Com o padrão CORBA-JSP-Gateway , estes problemas são resolvidos, visto que são tecnologias projetadas para o funcionamento com ou sem cliente/servidor e orientação a objetos.

CORBA e Java são tecnologias muito mais realizadoras do que parece. CORBA, além de um corretor de pedido de objeto (ORB), também é uma plataforma de objeto distribuído muito completa. CORBA estende o alcance de suas aplicações de Java por redes, linguagens, componentes e sistemas operacionais.

Por outro lado, Java também é mais do que uma linguagem com ligações com CORBA. Java é um sistema de objeto móvel; é um sistema operacional para objetos correntes. Java permite que seus objetos CORBA executem em mainframes para redes de computadores, telefones celulares, etc. Java simplifica a distribuição de código em sistema CORBA. Seus bytecodes possibilitam novos comportamentos para CORBA como agentes móveis. Hoje a linguagem Java parece ser a melhor opção para escrever clientes e servidores objetos CORBA. Seus multithreading embutidos, sua coleção de lixo e gerenciamento de erros torna mais fácil escrever objetos em rede robusta.

Analisando estas duas infra-estruturas, verifica-se que elas se complementam. Java começa onde CORBA termina. CORBA trabalha com a transparência de rede e Java trabalha com a transparência de implementação. CORBA provê o elo entre o ambiente Java de aplicação portátil e o mundo de objetos.

Logo, criou-se um ORB de CORBA/Java, o que nada mais é do que um ORB de CORBA/IIOP, o qual é escrito totalmente em Java, permitindo toda a portabilidade que é conhecida. Os principais ORBs que permitem este desenvolvimento no mercado são: Sun's Joe, Iona's OrbixWeb e Visigenic/Netscape's Visibroker for Java, sendo que cada um destes ORBs tem investidores fortes.

Para que a aplicação Balcão venha a ser utilizada via Internet utilizando o catálogo de padrões de projeto para aplicações distribuídas, o browser deverá executar uma página armazenada no servidor, escrita em HTML/JSP que mostramos a seguir:

A página JSP abaixo, tem como objetivo mostrar a figura 25 e executar as regras de negócio que fazem parte do seu funcionamento.

Página JSP

```
<html>
<%@ page language="java" import="aplicacao.balcao.*"%>
<jsp:useBean id="eventos" scope="request" class="aplicacao.balcao.EventosInterface"/>
<head>
	<title></title>
</head>

<body>
<%
  eventos.tratarEventos(request);
%>
<div align="center">
<form>
<table border="0" bgcolor="#000000" cellpadding="0" cellspacing="2"><tr><td>
<table border="0" width="500" cellpadding="5" cellspacing="0">
<tr>
	<td colspan="4" align="center" bgcolor="#000080"><font size="2" color="#FFFFFF"
face="Arial"><b>Objetos & Cia</b></font></td>
</tr>
<tr>
	<td bgcolor="#C0C0C0" align="center"><font size="1" face="Arial"><b>Código de
produto</b></font></td>
	<td bgcolor="#C0C0C0"><input type="text" name="codProd"></td>
	<td bgcolor="#C0C0C0" align="center"><font size="1"
face="Arial"><b>Quantidade</b></font></td>
	<td bgcolor="#C0C0C0"><input type="text" name="qtd"></td>
</tr>
<tr>
	<td bgcolor="#C0C0C0" align="center"><font size="1" face="Arial"><b>Total</b></font></td>
	<td bgcolor="#C0C0C0" colspan="3"><input type="text" name="total"
value="<%=eventos.getTotal()%>"></td>
</tr>
<tr>
	<td bgcolor="#C0C0C0" align="center"><font size="1"
face="Arial"><b>Fornecido</b></font></td>
	<td bgcolor="#C0C0C0"><input type="text" name="fornecido"
value="<%=request.getParameter("fornecido")%>"></td>
	<td bgcolor="#C0C0C0" align="center"><font size="1" face="Arial"><b>Troco</b></font></td>
	<td bgcolor="#C0C0C0"><input type="text" name="troco"
value="<%=eventos.getTroco()%>"></td>
</tr>
<tr bgcolor="#C0C0C0">
	<td colspan="4" align="center">
		<table border="0" cellpadding="3" cellspacing="0">
		<tr>
			<td><input type="submit" name="btEntraItem" value="Entrar Item"
style="width: 150px;"></td>
			<td><input type="submit" name="btTermVenda" value="Terminar Venda"
style="width: 150px;"></td>
			<td><input type="submit" name="btRegPag" value="Registrar Pagamento"
style="width: 150px;"></td>
		</tr>
		</table>
	</td>
</tr>
```

```
</table>
</td></tr></table>
</form>
</div>
</body>
</html>
```

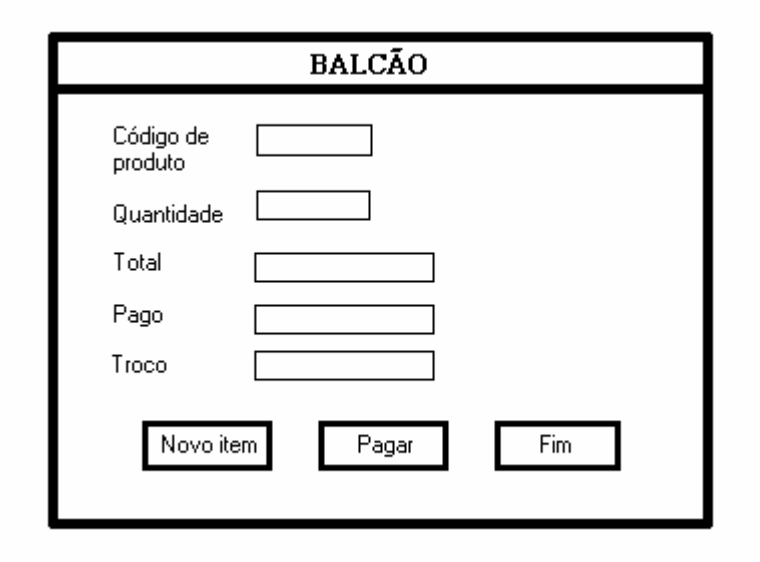

Figura 25 – Apresentação do sistema.

Como queremos aplicar o paradigma reutilização em toda a aplicação, criamos um componente chamado “EventosInterface”, para que a página escrita em JSP não tivesse classes implícitas. O código abaixo demonstra tal utilização.

Classe EventosInterface

```
package aplicacao.balcao;
import aplicacao.corba.*;
import javax.servlet.http.HttpServletRequest;
public class EventosInterface {

 //Atributos necessários para operação da Interface
 private Gateway gateway;
 private DIIGateway diigateway;
 private Balcao balcao;
 private float troco = 0;
 private float total = 0;
```

```
public EventosInterface() {
}
public void tratarEventos(HttpServletRequest request) {
    if (request.getParameter("fsNome") == null) {
        if (gateway == null) {
          gateway = new Gateway(request.getSession(true).getId());
        }
        balcao = gateway.getBalcao();
        total = 0;
        troco = 0;
        //Entra novo item de venda
        if (request.getParameter("btEntraItem") != null)
        {
            int upc = Integer.parseInt(request.getParameter("codProd"));
            int quantidade = Integer.parseInt(request.getParameter("quantidade"));
            balcao.entrarItem(upc, quantidade);
        } else
        //Finaliza a Venda
        if (request.getParameter("btTermVenda") != null)
        {
            balcao.terminarVenda(); //Termina a Venda
            total = balcao.getTotal(); //Pega o Total
        } else
        //Registra o Pagamento
        if (request.getParameter("btRegPag") != null)
        {
            float dinheiroFornecido = Float.parseFloat(request.getParameter("fornecido"));
            balcao.registrarPagamento(dinheiroFornecido); // Registra o Pagamento
            troco = balcao.getVenda().getTroco(balcao.getTotal()); // Pega o troco calculado
        }
    } else {
        if (diigateway == null) {
          diigateway = new DIIGateway(request.getSession(true).getId(), request.getParameter("fsNome"),
request.getParameter("wsNome"), request.getParameter("fsMetodo"), request.getParameter("wsMetodo"));
        }
        balcao = diigateway.getBalcao();
        total = 0;
        troco = 0;
        //Entra novo item de venda
        if (request.getParameter("btEntraItem") != null)
        {
            int upc = Integer.parseInt(request.getParameter("codProd"));
            int quantidade = Integer.parseInt(request.getParameter("quantidade"));
            balcao.entrarItem(upc, quantidade);
        } else
        //Finaliza a Venda
        if (request.getParameter("btTermVenda") != null)
        {
            balcao.terminarVenda(); //Termina a Venda
            total = balcao.getTotal(); //Pega o Total
        } else
        //Registra o Pagamento
        if (request.getParameter("btRegPag") != null)
        {
            float dinheiroFornecido = Float.parseFloat(request.getParameter("fornecido"));
            balcao.registrarPagamento(dinheiroFornecido); // Registra o Pagamento
            troco = balcao.getVenda().getTroco(balcao.getTotal()); // Pega o troco calculado
```

```
            }
        }
    }

    public float getTroco() {
        return troco;
    }

    public float getTotal() {
        return total;
    }
}
```

Logo, para atender aos novos requisitos que exigem conversações altamente iterativas e distribuídas, com velocidade na geração de novo processo sem a necessidade de memória residente, utilizaremos o padrão CORBA-JSP-Gateway.

A arquitetura CORBA de sistemas distribuídos utilizado aqui é baseada em um pedido de serviço para um objeto distribuído. Para este paradigma, a arquitetura de CORBA tem as seguintes metas: abstração, localização com acesso transparente e interoperabilidade.

A abstração está na especificação de uma linguagem neutra IDL – Linguagem de definição de interfaces. A localização e meios de transparência de acesso diz respeito a invocação de um serviço sem o conhecimento de sua localização que é implementada por um ORB, o qual localiza o objeto remoto, comunica o pedido a este objeto e espera pelo resultado para ser enviado de volta ao cliente. A interoperabilidade ocorre em nível de sistema operacional, de rede, arquitetura de computador e linguagem de programação. Para tanto, um protocolo de rede (IIOP) tem que apoiar o conjunto de formatos das mensagens e dados para a comunicação entre ORBs (GIOP).

Como iremos fazer uso da especificação CORBA, através do ORB do Visibroker, a seguinte arquitetura se faz necessária: através da linguagem de definição de interface (IDL), que é fundamental para a interoperabilidade de linguagens em CORBA, oferece construtores para descrever módulos, interfaces, métodos, exceções e tipos de dados que a seguir, são apresentadas no formato para aplicação Balcão, utilizada no padrão Gateway.

IDL Balcao

```
module aplicacao {
  module balcao {
   extensible struct Loja {
   };
   extensible struct Balcao {
   };
  };
  module corba {
     interface IWebServer {
       ::aplicacao::balcao::Balcao getBalcao(in string id);
       void putBalcao(
         in string id,
         in ::aplicacao::balcao::Balcao balcao
       );
      void removeBalcao(in string id);
     };
     interface IFactoryServer {
       ::aplicacao::balcao::Loja criaLoja();
       //::aplicacao::balcao::Balcao criaNovoBalcao();
     };
  };
};
```

O cliente de um serviço pode invocar dinamicamente com DII ou estaticamente com Stubs de IDL. Na invocação dinâmica, o cliente constrói o serviço chamado em tempo de execução, não necessitando de um Stub. Este método de programação é flexível, porém, muito mais complexo que a invocação estática. Logo, a invocação estática é a mais utilizada, devido a sua geração dar-se através de uma pré-compilação da interface.

Na invocação estática, o cliente inicializa o ORB, usa o serviço de nome para obter a referência do objeto remoto e interage com este por meio da interface. O código seguinte mostra estes passos:

Padrão CORBA-JSP-Gateway

```
package aplicacao.corba;

import aplicacao.balcao.*;
import org.omg.CORBA.ORB;
public final class Gateway {
 private Loja loja;
 private Balcao balcao;
 private IFactoryServer fserver;
 private IWebServer wserver;
 private ORB orb;
 public Gateway(String id) {
   this(id, new String[]{});
 }
 public Gateway(String id, String[] args) {
   orb = ORB.init(args,null);
   fserver = IFactoryServerHelper.bind(orb, "FactoryServer");
   wserver = IWebServerHelper.bind(orb, "WebServer");
   loja = fserver.criaLoja();
   setBalcao(id, loja.getBalcao());
   System.out.println(fserver._object_name() + " foi criado com sucesso.");
   System.out.println(wserver._object_name() + " foi criado com sucesso.");
   System.out.println(wserver.getBalcao(id) + " foi criado com sucesso.");
 }
 public Loja getLoja() {
    return loja;
 }

 public void setLoja(Loja lj) {
     loja = lj;
 }
 public Balcao getBalcao() {
```

```
    return balcao;
  }
  public void setBalcao(String id, Balcao b) {
     balcao = b;
     wserver.putBalcao(id, loja.getBalcao());
  }
  public static void main(String[] args) {
     Gateway client = new Gateway("1", args);
  }
}
```

O objeto remoto também pode ser implementado para suportar a invocação estática ou dinâmica. Como mencionado anteriormente, a invocação estática tem suas vantagens, sendo por isso mais utilizada. Com a invocação estática, o objeto remoto que implementa o serviço tem que estender o skeleton, o qual também é gerado ao compilar o método de interface. O código abaixo mostra a implementação de um objeto remoto:

FactoryServer

```
package aplicacao.corba;

import aplicacao.balcao.*;
public final class FactoryServer extends _IFactoryServerImplBase {
  org.omg.CORBA.ORB orb;
  org.omg.CORBA.BOA boa;
  public FactoryServer(String name) {
     super(name);
  }
  public void esperaPorConexao(String[] args) {
    orb = org.omg.CORBA.ORB.init(args,null);

    boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
    boa.obj_is_ready(this);
    System.out.println(this + " esta pronto.");
    boa.impl_is_ready();
  }
```

```
  public Loja criaLoja() {
     return Loja.instancia();
  }
  }
```

WebServer

```
package aplicacao.corba;

import aplicacao.balcao.*;
import java.util.Hashtable;
public class WebServer extends _IWebServerImplBase {
  private String _nome;
  private Hashtable htBalcao = new Hashtable();
  public WebServer(String nome) {
      super(nome);
      _nome = nome;
  }
  public void esperaPorConexao(String[] args) {
     org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
     org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
     boa.obj_is_ready(this);
     System.out.println(this + " esta pronto.");
     boa.impl_is_ready();
  }
  public Balcao getBalcao(String id) {
      return (Balcao) htBalcao.get(id);
  }
  public void putBalcao(String id, Balcao balcao) {
      htBalcao.put(id, balcao);
  }
  public void removeBalcao(String id) {
      htBalcao.remove(id);
  }
}
```

Por fim, inicializa-se o ORB no servidor e cria-se um objeto remoto que é conectado ao ORB que é registrado ao serviço de nome, ficando a espera de invocações. Abaixo, as classes que devem estar presentes no servidor.

ServidorFactory

```
package aplicacao.corba;
public class ServidorFactory {
 public static void main(String[] args) {
     FactoryServer  server = new FactoryServer("FactoryServer");
     server.esperaPorConexao(args);
  }
}
```

ServidorWeb

```
package aplicacao.corba;
public class ServidorWeb {
  public static void main(String[] args) {
     WebServer  server = new WebServer("WebServer");
     server.esperaPorConexao(args);
  }
}
```

No processo anteriormente citado está presente o padrão Gateway que provê dois benefícios imediatos:

- CORBA evita o gargalo de CGI, permitindo aos clientes invocar métodos diretamente em um servidor. O cliente passa os parâmetros diretamente por stubs e o servidor recebe a chamada por um skeleton pré-compilado;

- CORBA provê uma infra-estrutura de servidor para servidor. Estes objetos podem correr em servidores múltiplos, provendo balanceamento de carga para pedidos de clientes.

O padrão CORBA-JSP-Gateway não é difícil de ser utilizado, bastando definir uma classe que representa o ponto de partida de um sistema, sendo que ele geralmente é representado em sistemas legados através de um programa menu.

Este padrão garante um bom controle quanto ao gerenciamento de complexidade, mudanças e extensibilidade.

Para tentar absorver todo o processo dinâmico que a WEB vem necessitando obter, em virtude do crescimento dos negócios e usuários, frente as novas tecnologias, o padrão CORBA-Web-Server, vem beneficiar através de retenção da informação do estado sobre uma aplicação de servidor, e também melhorando o desempenho, através da limitação do tamanho de endereços URL.

Quando um objeto é instanciado, seu endereço é convertido para um símbolo, um label ou um nome qualquer para identificar o objeto atual, sendo que este símbolo será passado como uma variável fixa na URL.

O que significa dizer que, em um eventual acesso a um cliente CORBA, pode haver a necessidade de alguma modificação em seu estado do objeto, onde, em vez de pegar uma lista de variáveis escondidas no fim da URL para representar o seu estado atual, simplesmente envia o símbolo para o objeto.

Sempre que este cliente CORBA for invocado, o símbolo será usado para identificar o objeto CORBA atual e sua referência de objeto será retornada pelo serviço de nome da OMG.

Com este processo, o ambiente WEB, só necessitará armazenar o símbolo do objeto, podendo ser utilizado como criação dinâmica de uma página de HTML, provendo um aumento de portabilidade dos URLs. A classe abaixo mostra a aplicação do padrão CORBA-Web-Server.

Classe WebServer

```
package aplicacao.corba;

import aplicacao.balcao.*;
import java.util.Hashtable;
public class WebServer extends _IWebServerImplBase {
 private String _nome;
 private Hashtable htBalcao = new Hashtable();
 public WebServer(String nome) {
    super(nome);
    _nome = nome;
 }
 public void esperaPorConexao(String[] args) {
   org.omg.CORBA.ORB orb = org.omg.CORBA.ORB.init(args,null);
   org.omg.CORBA.BOA boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
   boa.obj_is_ready(this);
   System.out.println(this + " esta pronto.");
   boa.impl_is_ready();
 }
 public Balcao getBalcao(String id) {
    return (Balcao) htBalcao.get(id);
 }
 public void putBalcao(String id, Balcao balcao) {
    htBalcao.put(id, balcao);
 }
 public void removeBalcao(String id) {
    htBalcao.remove(id);
 }
 public static void main(String[] args) {
    WebServer  server = new WebServer("WebServer");
    server.esperaPorConexao(args);
 }
}
```

Na realidade, este padrão mostra mais sua eficiência em um projeto que necessite de inúmeras telas sobrepostas, ou seja, para poder ir voltando telas como usuário de Internet, faz-se necessário dar a localização de cada página(tela do sistema).

Este padrão não oferece preocupação pois a tabela na qual são armazenados os endereços e seus símbolos são de fácil acesso e armazenamento.

Este padrão deve ser utilizado em todos os projetos.

O padrão factory tem como intenção definir uma interface para criar um objeto, mas que também deixe as subclasses decidirem em que classe instanciar, ou seja, como podemos ter vários balcões em uma mesma loja, queremos que todas utilizem as mesmas informações gerais lidas apenas uma vez. Na nossa aplicação este padrão vem junto a necessidade de distribuição dos objetos. A classe Gateway, quando acessada utilizando o padrão CORBA-JSP-Gateway, irá instanciar a aplicação Balcão através da classe Loja. O padrão factory elimina a necessidade de se anexar classes específicas das aplicações no código, ou seja, o código somente lida com a interface gateway. Em geral, a fábrica de objetos possui métodos com listas de parâmetros que espelham a lista do construtor.

Este padrão é aplicado em ambiente distribuído, onde instanciar objetos remotos são necessários. A classe abaixo mostra como fazer para instanciar um objeto remoto sem necessitar da palavra-chave para localizar um objeto em uma máquina remota.

O ambiente CORBA, fazendo uso dos pedidos de correção para localizar um objeto remoto, vem prover interoperabilidade entre ambientes heterogêneos, nos quais clientes simultâneos múltiplos podem fazer uso do padrão factory em threads seguros, onde, em havendo demora em uma instanciação de um objeto, a resposta de cada pedido será em um thread separado.

Classe Balcao

```
package aplicacao.corba;
import aplicacao.balcao.*;
public final class FactoryServer extends _IFactoryServerImplBase {
 org.omg.CORBA.ORB orb;
 org.omg.CORBA.BOA boa;

 Loja loja;
 public FactoryServer(String name) {
    super(name);
 }
 public void esperaPorConexao(String[] args) {
   orb = org.omg.CORBA.ORB.init(args,null);
   boa = ((com.visigenic.vbroker.orb.ORB)orb).BOA_init();
   boa.obj_is_ready(this);
   System.out.println(this + " esta pronto.");
   boa.impl_is_ready();
 }
 public Loja criaLoja() {
   if (loja == null) {
     System.out.println("Nova Loja");
     loja = new Loja();
   }
   return loja;
 }
 public static void main(String[] args) {
    FactoryServer  server = new FactoryServer("FactoryServer");
    server.esperaPorConexao(args);
 }
}
```

Na aplicação Balcão iremos inserir na classe loja o padrão Singleton que terá como função prover uma instância única nesta classe. O objetivo é garantir que outras lojas que forem criadas tenham o mesmo endereço e o mesmo conteúdo de informação torna-se também fácil o controle de quantas lojas forem criadas.

Classe Loja

```
package aplicacao.balcao;
import java.io.Serializable;
public class Loja implements Serializable{
  //Padrao Singleton
  private static Loja instancia;

  public Loja() {
  }
  private CatalogodeProdutos catalogodeProdutos = new CatalogodeProdutos();
  public Balcao getBalcao() {
    Balcao balcao = new Balcao(catalogodeProdutos);
    return balcao;
  }
  //Padrao Singleton
  public static Loja instancia() {
     if (instancia == null)
        instancia = new Loja();
     return instancia;
  }
}
```

Este padrão é simples e pode ser utilizado em objetos.

Um outro padrão que iremos utilizar é o padrão State, que pode ser utilizado para eliminar testes de condições que são causadas por dependências de estado. Na aplicação temos que diferenciar quando uma venda é nova ou não. Logo, devemos criar classes para cada estado que influencia o comportamento do objeto, dependendo do estado, ou seja, baseado no polimorfismo, criam-se métodos para cada classe que representa um estado para tratar o comportamento do objeto. A seguir, mostramos as classes implementadas pelo padrão.

Classe Balcao

```
package aplicacao.balcao;
import java.util.*;
import java.io.Serializable;
public class Balcao implements Serializable
{
 private CatalogodeProdutos catalogodeProdutos;
 private Venda venda;
 //Padrao State
 private EstadoBalcao estado;
 private float total;
 public Balcao(CatalogodeProdutos catalogo)
 {
    catalogodeProdutos = catalogo;
    catalogodeProdutos.regListener(this);
    estado = new EstadoEsperandoPorVenda(this);
 }
 public void terminarVenda()
 {
    venda.tornaseCompleta();
    calcularTotal();
    catalogodeProdutos.atualizaEstoque(venda);
    estado = new EstadoEsperandoPorVenda(this);
 }
 public void calcularTotal()
 {
   venda.totaliza(this);  //Cliente callback
 }
 //Metodo chamado pelo servidor callback
 public void totalCalculado(float tot)
 {
    total = tot;
 }
 public void entrarItem(int upc, int quantidade)
 {
   estado.entrarItem(upc, quantidade);  }
```

```
public void registrarPagamento(float dinheiroFornecido)
{
   venda.efetuarPagamento(dinheiroFornecido);
}
public void mudouEstoque(CatalogodeProdutos catalogo) {
   catalogodeProdutos = catalogo;
}
public Venda getVenda()
{
    return venda;
}
public EstadoBalcao getEstado()
{
    return estado;
}
public CatalogodeProdutos getCatalogo()
{
    return catalogodeProdutos;
}
public void setVenda(Venda v)
{
    venda = v;
}
public void setEstado(EstadoBalcao e)
{
    estado = e;
}}
```

Classe EstadoBalcao

```
package aplicacao.balcao;

public abstract class EstadoBalcao {
 Balcao b;
 public abstract void entrarItem(int upc, int quantidade);
}
```

Classe EstadoEntradocomVenda

```
package aplicacao.balcao;
public class EstadoEntrandocomVenda extends EstadoBalcao {
  public EstadoEntrandocomVenda(Balcao balcao) {
     b = balcao;
  }
  public void entrarItem(int upc, int quantidade)
  {
     Venda v = b.getVenda();
     EspecificacaodeProduto espec = b.getCatalogo().getEspecificacao(upc);
     v.construirLinhadeItem(espec, quantidade);
  }
}
```

Classe EstadoEsperandoPorVenda

```
package aplicacao.balcao;
public class EstadoEsperandoPorVenda extends EstadoBalcao {
  public EstadoEsperandoPorVenda(Balcao balcao) {
     b = balcao;
  }
  public void entrarItem(int upc, int quantidade)
  {
     Venda v = new Venda();
     EspecificacaodeProduto espec = b.getCatalogo().getEspecificacao(upc);
     v.construirLinhadeItem(espec, quantidade);
     b.setVenda(v);
     b.setEstado(new EstadoEntrandocomVenda(b));
  }
}
```

O padrão state é um dos mais fáceis de saber onde deve ser implementado, pois basta que haja tipos de condições diferentes para um mesmo processo para poder ser utilizado.

Este padrão também minimiza a complexidade e a extensibilidade do sistema, pois para cada condição nova, basta criar uma nova classe.

Na nossa implementação não houve, porém, o uso do teste de condição pois, quando executada uma determinada classe, uma palavra chave é utilizada para mudar a condição do seu estado. Em havendo mais do que duas condições, é provável que o teste seja inevitável.

Este padrão deve ser utilizado para tornar a programação mais visível e de fácil entendimento.

Em nossa aplicação, poderíamos incrementar o uso do padrão state quanto a forma de pagamento: poderíamos acrescentar o pagamento em cheque, pagamento com cartão de crédito ou pagamento em dinheiro.

Em qualquer aplicação, podem haver processos que levam tempo demais para ser executados, comprometendo a execução como um todo. O padrão Callback, reduz o tempo de espera que um cliente necessitaria ter, em uma consulta em servidor, mas não para um processo qualquer, onde esta operação seria convertida de síncrona para assíncrona. Utilizando o padrão callback, o cliente pode executar outras tarefas enquanto não obtiver o resultado de uma solicitação ao servidor.

Este padrão de projeto é aplicado em um meio onde o processamento do servidor requer um tempo de resposta grande que, neste caso, pode não ficar bem configurada tal necessidade, mas que inserimos para a utilização do método. Este padrão pode ser de grande valia em computação distribuída, liberando o cliente para executar outras tarefas enquanto espera o retorno solicitado.

Classe Venda

```
package aplicacao.balcao;
import java.util.*;
import java.io.Serializable;
public class Venda implements Serializable {
  private Vector linhadeItens = new Vector();
  private Date data = new Date();
  private boolean estaCompleta = false;
  private Pagamento pagamento;

  public float getTroco(int total) {
     return pagamento.getQuantia() - total;
  }
  public void tornaseCompleta() {
     estaCompleta = true;
  }
  public boolean estaCompleta() {
     return estaCompleta;
  }
  public void construirLinhadeItem (EspecificacaodeProduto espec, int quantidade) {
     linhadeItens.addElement(new LinhadeItemdeVenda(espec, quantidade));
  }
  public Vector getLinhadeItens() {
     return linhadeItens;
  }
  public void totaliza(Balcao cliente) {

     // Padrão Callback
     ProcessoCallback processo = new ProcessoCallback(cliente);
     processo.start();
  }
  public void efetuarPagamento(float dinheiroFornecido) {
     pagamento = new Pagamento(dinheiroFornecido);
  }
  class ProcessoCallback extends Thread {
     private Balcao _cliente;
     public ProcessoCallback(Balcao cliente) {
        _cliente = cliente;
```

```
    }
    public void run() {
      float total = 0;
      Enumeration e = linhadeItens.elements();
      while (e.hasMoreElements()) {
        total += ((LinhadeItemdeVenda) e.nextElement()).subtotal();
      }
      _cliente.totalCalculado(total);
    }
  }
}
```

Para a implantação deste padrão, é preciso ter certeza de que o resultado deste processo não seja necessário na continuação do mesmo. Geralmente, é mais utilizado em rotinas (classes) que trabalham com emissão de relatório, liberando a máquina para outros processos.

O padrão observer tem como funcionalidade constatar mudanças de estado em objetos servidores e notificar aos objetos clientes sobre a atualização das mudanças ocorridas, sendo este processo uma exigência comum em sistemas distribuídos.

Ocorrências de mudanças de estado entre objetos cliente e servidor são freqüentes. Logo, conferências de mudanças podem requerer o consumo de recursos de clientes, recursos de servidor e bandwidt (banda larga).

O padrão observer garantirá que as atualizações de mudanças entre objetos clientes e servidores sejam ouvidas sempre que necessárias. Em havendo dispositivos de segurança externos (firewall), estes devem ser conhecidos para que os métodos de invocação possam ser utilizados.

Classe Balcao

```
package aplicacao.balcao;
import java.util.*;
import java.io.Serializable;
public class Balcao implements Serializable
```

```
{
 private CatalogodeProdutos catalogodeProdutos;
 private Venda venda;
 //Padrao State
 private EstadoBalcao estado;
 private float total;
 public Balcao(CatalogodeProdutos catalogo)
 {
    catalogodeProdutos = catalogo;
    catalogodeProdutos.regListener(this);
    estado = new EstadoEsperandoPorVenda(this);
 }
 public void terminarVenda()
 {
    venda.tornaseCompleta();
    calcularTotal();
    catalogodeProdutos.atualizaEstoque(venda);
    estado = new EstadoEsperandoPorVenda(this);
 }
 public void calcularTotal()
 {
   venda.totaliza(this);  //Cliente callback
 }
 //Metodo chamado pelo servidor callback
 public void totalCalculado(float tot)
 {
    total = tot;
 }
 public void entrarItem(int upc, int quantidade)
 {
   estado.entrarItem(upc, quantidade);
 }
 public void registrarPagamento(float dinheiroFornecido)
 {
   venda.efetuarPagamento(dinheiroFornecido);
 }
 public void mudouEstoque(CatalogodeProdutos catalogo) {
   catalogodeProdutos = catalogo;
```

```
  }
  public Venda getVenda()
  {
     return venda;
  }
  public EstadoBalcao getEstado()
  {
     return estado;
  }
  public CatalogodeProdutos getCatalogo()
  {
     return catalogodeProdutos;
  }
  public void setVenda(Venda v)
  {
     venda = v;
  }
  public void setEstado(EstadoBalcao e)
  {
     estado = e;
  }
}
```

Classe CatalogodeProdutos

```
package aplicacao.balcao;
import java.util.*;
import java.io.Serializable;
public class CatalogodeProdutos implements Serializable{
  private Hashtable especificacoesdeProdutos = new Hashtable();
  private Vector vecListeners = new Vector();
  public CatalogodeProdutos() {
     EspecificacaodeProduto ep = new EspecificacaodeProduto( 100, 1, "PRODUTO 1", 10);
     especificacoesdeProdutos.put(new Integer(100), ep);
     ep = new EspecificacaodeProduto( 200, 1, "PRODUTO 2", 10);
     especificacoesdeProdutos.put(new Integer(200), ep);
  }
```

```
  public EspecificacaodeProduto getEspecificacao(int upc) {
     return (EspecificacaodeProduto) especificacoesdeProdutos.get(new Integer(upc));
  }

  public void regListener(Balcao b) {
     if (!vecListeners.contains(b))
       vecListeners.addElement(b);
  }
  public void atualizaEstoque(Venda v) {
    Vector itens = v.getLinhadeItens();
    for (int i=0; i < itens.size();i++) {
        LinhadeItemdeVenda item = (LinhadeItemdeVenda) itens.elementAt(i);
        EspecificacaodeProduto espec= item.getEspecdeProduto();
        espec.setQuantidade(espec.getQuantidade() - item.getQuantidade());
        especificacoesdeProdutos.put(new Integer(espec.getUPC()), espec);
    }
    Enumeration listeners = vecListeners.elements();
    while (listeners.hasMoreElements()) {
        ((Balcao)listeners.nextElement()).mudouEstoque(this);
    }
  }
}
```

5.6. Utilizando o padrão dinâmico DII-Web-Server.

O padrão DII-Web-Server tem como objetivo executar as classes distribuídas da mesma forma que o padrão CORBA-JSP-Gateway. DII-Web-Server utilizará os conceitos dinâmicos da especificação CORBA, enquanto que o padrão CORBA-JSP-Gateway, utiliza o processo estático.

O padrão DII-Web-Server utiliza os princípios do CORBA dinâmico que permite escolher qual das suas classes vão ser executadas.

Como a pagina JSP foi projetada para que pudesse ser reutilizada tanto no padrão estático como no dinâmico, não necessitará mudanças.

Para o padrão DII-Web-Server, deverá ser incrementada uma interface antes da execução da página JSP principal, através de uma página HTML, que solicitará quais as classes e métodos que se quer executar, como mostra a figura 26.

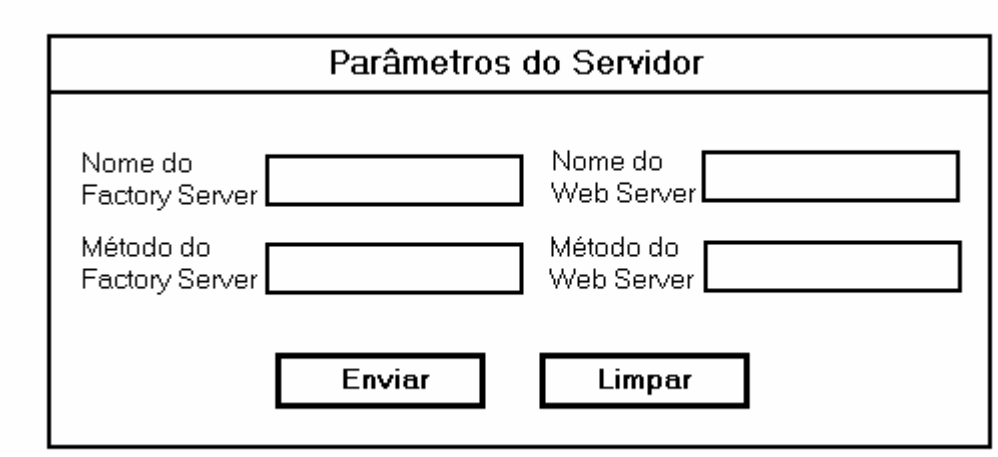

Figura 26 – Inicializando sistema pelo padrão DII-Web-Server.

Esta interface irá servir de entrada dos parâmetros para a execução das classes FactoryServer e WebServer dinamicamente, utilizando a mesma página JSP utilizada pelo padrão estático com o objetivo de se executar a aplicação balcão.

Para finalizar o padrão DII-Web-Server, terá sua classe cliente alterada para DIIGateway e manterá inalteradas as classes do servidor do padrão CORBA-JSP-Gateway.

Classe DIIGateway

```
package aplicacao.corba;
import aplicacao.balcao.*;
import org.omg.CORBA.ORB;
public final class DIIGateway {
  private Loja loja;
  private Balcao balcao;
  private IFactoryServer fserver;
  private IWebServer wserver;
  private ORB orb;

  public DIIGateway(String id, String fServerNome, String wServerNome, String ServerMetodo, String
wServerMetodo) {
     orb = ORB.init(new String[]{},null);
     fserver = IFactoryServerHelper.bind(orb, fServerNome);
     org.omg.CORBA.Request request = fserver._request(fServerMetodo);
     request.set_return_type(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_objref));
     request.invoke();
     loja = (Loja)request.return_value().extract_Object();
     balcao = loja.getBalcao();
     wserver = IWebServerHelper.bind(orb, wServerNome);
     request = wserver._request(wServerMetodo);
     request.add_in_arg().insert_string(id);
     request.add_in_arg().insert_TypeCode(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_objref));
     request.set_return_type(orb.get_primitive_tc(org.omg.CORBA.TCKind.tk_void));
     request.invoke();
     System.out.println(fserver._object_name() + " foi criado com sucesso.");
     System.out.println(wserver._object_name() + " foi criado com sucesso.");
     System.out.println(wserver.getBalcao(id) + " foi criado com sucesso.");
  }

  public Loja getLoja() {
     return loja;
  }

  public void setLoja(Loja lj) {
     loja = lj;
  }

  public Balcao getBalcao() {
     return balcao;
```

```
  }

  public static void main(String[] args) {
    DIIGateway diig = new DIIGateway("1", "FactoryServer", "WebServer", "criaLoja", "putBalcao");
  }
}
```

6. CONCLUSÕES

Os padrões de projeto não apresentam nenhum algoritmo ou técnica de programação que já não se tenha utilizado e nem mesmo fornecem um método rigoroso para projetar sistemas. Mas, eles permitem a criação de modelos que podem ser utilizados tanto como documentação, quanto como códigos fontes, por desenvolvedores novos ou com experiência para melhorar seus trabalhos presentes e futuros. Em conseqüência ocorrerá uma melhora da qualidade dos softwares desenvolvidos através da reutilização de seus processos.

Esta dissertação apresenta um catálogo de padrões de projeto para aplicações distribuídas onde, em um estudo de caso, mostramos sua aplicação para a implantação em ambiente WEB ou em outras estruturas organizacionais tais como, um setor bancário ou uma loja qualquer.

As principais contribuições deste trabalho foram: a criação de um catálogo de padrões de projeto para aplicações distribuídas com CORBA, garantindo a interoperabilidade entre sistemas; a implementação, visando objetos Web; fornecer exemplos escritos em linguagem de programação Java, permitindo portabilidade e direcionamento para estruturas organizacionais, como nos setores bancário, médico e para os desenvolvedores de softwares em geral. Isto significa colocar a disposição de todos, elementos que possam ser reutilizados.

Dois princípios fundamentais devem ser levados em conta quando se pensa em padrões de projeto:

1. Padrões de projeto são um ponto de partida, onde deve-se procurar soluções para seus problemas. Não são um ponto de destino, ou seja, não se deve desenvolver aplicativos pensando em criar padrões. Eles vêm em conseqüência de sua experiência;

2. Modelos criados não são certos ou errados: são simplesmente uma contribuição para futuros melhores, ou seja: o ponto de vista de cada indivíduo afeta o que vem a ser ou não um padrão.

Através do estudo de caso, podemos destacar os seguintes pontos:

- Os padrões de projeto propostos por esta dissertação atende amplamente a transição de um sistema legado para um sistema Web distribuído. Atende também a portabilidade, extensibilidade, interoperabilidade e reutilização de processos através das tecnologias utilizadas como CORBA e Java.

- Nota-se que todos os padrões de projeto foram tirados das características pertinentes as tecnologias utilizadas CORBA e Java.

- É necessário dizer também que padrões de projeto não estão vinculados a tecnologias específicas ou seja, independente das tecnologias utilizadas nas empresas de desenvolvimento, pode-se criar padrões de acordo com as suas características.

- A documentação estabelecida pelos elementos de padrões de projeto realmente conduzem a um vocabulário comum entre desenvolvedores dentro de uma empresa e tornam o aprendizado de novatos mais fácil.

- Introduzir a utilização de padrões de projeto dentro de uma empresa e fazer com que, as equipes de desenvolvimento utilizem e, busquem acrescentar novos padrões ao catálogo é o problema. Geralmente as empresas trabalham com um cronograma de desenvolvimento que impede que seus desenvolvedores tenham tempo de utilizar tal técnica.

- A popularidade atual da linguagem de programação Java principalmente em desenvolvimento para internet através de Applets, Servelets e JSP lhes conferindo alta flexibilidade, representada por uma linguagem orientada a objetos, multiplataforma, multithreading e com forma de acesso a bases de dados padronizada.

Os trabalhos futuros que podemos destacar são:

- Implementação dos padrões de projetos constante desta dissertação em ferramentas para auxilio no desenvolvimento de projetos de softwares.

- É um alvo para refatoração, ou seja, freqüentemente o software tem que ser reorganizado ou refatorado. Os padrões de projeto ajudam a determinar como reduzir o volume de refatoração.

7. REFERÊNCIAS BIBLIOGRÁFICAS

[ALEX97] Alexander C., Ishikawa S., Silverstein M., Lacobson M., Fiksdahl-King I., Angel S.. A Pattern Language. Oxford University Press, New York, 1997.

[BELL01] Belloquim, Átila. Reutilização de Objetos: Promessao ou Dívida? Developer's Magazine. 2001.

[BERN96] Bernhardt, Martin. Design and Implementation of a Web-based Tool for ATM Connection Management. Stuttgart, Aug. 1996.

[BETZ95] Betz, Mark. OMG's CORBA. Objects. Dr. Dobb's, San Mateo, V19, n16, p.8-12, winter, 1995.

[BILL98] Bill McCarty, et al. Java Distributed Objects. Sams 1998.

[BOOC92] Booch, Grandy. The Booch Method: Notation Part I. Computer Language, September – 1992.

[BOOC94] Booch, G.. Object-Oriented Analysis and Design with Applications. Benjamin/ Cummings, Redwood City, CA, 1994. Segunda Edição.

[BUSC96] Buschmann, F. System of Patterns: Pattern-Oriented Software Architecture. John Wiley & Sons, 1996.

[CARG92] Cargil, Tom. C++ Constructing Style. Addison-Wesley, Reading, MA 1992.

[COAD92] Coad, P., Yourdon, E.. Análise Baseada em Objetos. Editora Campus/ Yourdon Press. 1992.

[CORBA02] CORBA. www.omg.org.

[COUL95] Couloris, G., Dollimore J., Kindberg, T., Distribuited Systems: concepts and Design. Addison-Wesley, 1995.

[ERIC00] Erich G., Helm R., Johnson R., Vlissides J.. Padrões de Projeto: Soluções Reutilizáveis de Software Orientado a Objetos. Trad. Luiz A . M. Salgado. Bookman, 2000.

[GEMS97] Gemstone. Gemstone Role in CORBA Systems, white paper, www.gemstone.com, 1997.

[JACO92] Jacobson, I., Christerson, M., Jonsson, P., Overgaard, G.. Object-Oriented Software Engineering – A Use Case Driven Approach. Addison-Wesley, Workingham, 1992.

[JAVA02] Java. www.java.sun.com.

[KHOS86] Khoshafian, S. & Copeland, G. Object Identify SIGPLAN Notives, Vol. 21 (11). 1986.

[LARM00] Larman Craig. Utilizando UML e Padrões. 2000.

[LINT92] Linton, M., Calder, P., Interrante, J., Tang, S., Vlissides, J.. Inter Views Reference Manual. CSL, Stanford University, 3.1 edition, 1992.

[MAIN98] Mainetti Sergio. Artigo Objetos Distribuídos, 1998.

[MOWB97] Mowbray, T.J., Malveau, R.C.. CORBA Design Patterns. New York, NY: John Whiley and Sons, Inc. 1997.

[OMG97] Object Management Group. Common Object Service Specification. 1997.

[ORFA96] Orfali, R., Harkey, D., Edwards, J.. Essential Client/Server Survival Guide – Second Edition. New York, NY: John Wiley and Sons, Inc. 1996.

[ORFA97] Orfali, R., Harkey, D.. Client/Server Programming with Java and CORBA. New York, NY: John Wiley and Sons Inc. 1997.

[ORFA95] Orfali, R., Harkey, D., Edwards, J.. Intergalactic Client/Server Computing. In: Byte Vol. 20, No. 4. New York, NY: McGraw-Hill, April. 1995.

[OTTE96] Otte, Randy, Patrik, Paul, Roy, Mark. Understanding CORBA – The common Object Request Broker Architecture. New Jersey: Prentice-Hall, 1996.

[RATI 2] Rational Rose. www.rational.com.

[RICC00] Riccioni, Paulo. Introdução a Objetos Distribuídos com CORBA. Visual Books, 2000.

[RUMB91] Rumbaugh, L., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W.. Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, NJ, 1991.

[SEYB00] Seybold, Patrícia B.; Marshak, Roni T. Clientes.com: como criar uma estratégia empresarial para a Internet que proporcione lucros reais. São Paulo: Macron Books, 2000.

[TOGE02] TogetherSoft. www.togethersoft.com.

[VISI97] Visigenic Software, Inc. Distributed Object Computing in the internet Age, www.visigenic.com, 1997.